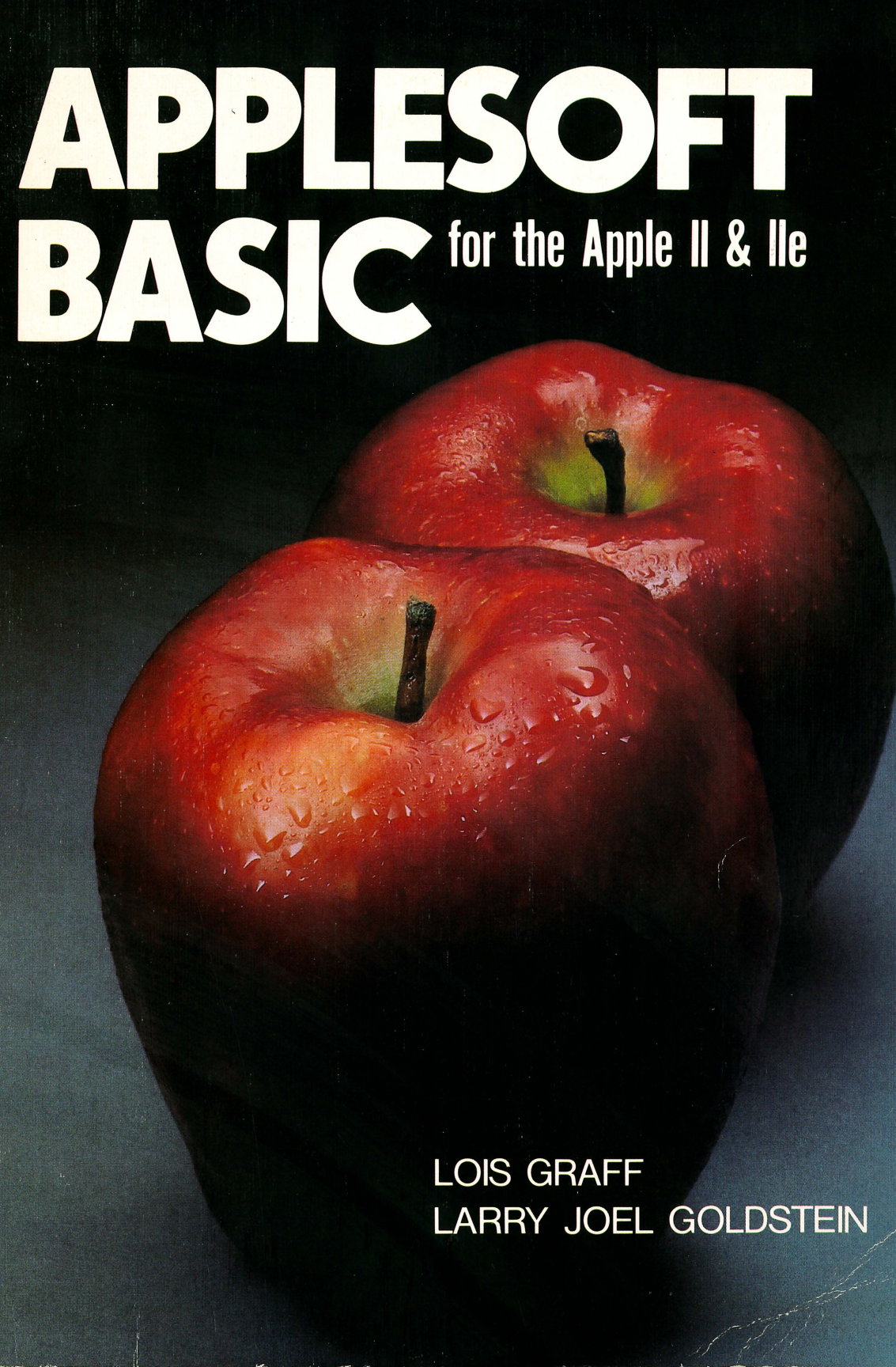
APPLESOFT BASIC

for the Apple II & IIe



LOIS GRAFF
LARRY JOEL GOLDSTEIN

Applesoft BASIC for the Apple II and IIe

Lois E. Graff
Larry Joel Goldstein

Robert J. Brady Co.
A Prentice-Hall Publishing and Communications Company
Bowie, MD 20715

Applesoft BASIC for the Apple II and IIe

Copyright © 1984 by Robert J. Brady Company.
All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage and retrieval system, without permission in writing from the publisher. For information, address Robert J. Brady Co., Bowie, Maryland 20715

Library of Congress Cataloging in Publication Data
Graff, Lois E., 1943-
Applesoft BASIC for the Apple II and IIe.

Includes index.
1. Apple II Plus (Computer)—Programming. 2. Apple IIe (Computer)—Programming. 3. Basic (Computer program language) I. Goldstein, Larry Joel. II. Title.
QA76.8.A6624G7 1983 001.64'2 83-15527
ISBN 0-89303-320-0

Prentice-Hall International, Inc., London
Prentice-Hall Canada, Inc., Scarborough, Ontario
Prentice-Hall of Australia, Pty., Ltd., Sydney
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Prentice-Hall of Southeast Asia Pte. Ltd., Singapore
Whitehall Books, Limited, Petone, New Zealand
Editora Prentice-Hall Do Brasil LTDA., Rio de Janeiro

Printed in the United States of America

84 85 86 87 88 89 90 91 92 93 94 10 9 8 7 6 5 4 3 2

Executive Editor: David T. Culverwell
Production Editor/Text Designer: Michael J. Rogers
Art Director/Cover Design: Don Sellers
Assistant Art Director: Bernard Vervin
Photography: George Dodson
Typefaces: Palatino (text), Folio (display), and OCR-B (computer programs)
Typesetting: Alexander Typesetting, Inc., Indianapolis, IN
Printer: Fairfield Graphics, Fairfield, PA
Indexing: Elliot Linzer

CONTENTS

Preface vi

1. A First Look at Computers 1
1.1 Introduction 1
1.2 What is a Computer? 4
1.3 Using Your Apple II Plus/IIe Computer Keyboard 7
1.4 Inside Your Apple 13
1.5 Using the Apple II Plus/IIe Disk Drive 21

2. Running Programs on Your Apple II Plus/IIe Computer 27
2.1 Computer Languages and Programs 27
2.2 Entering a Program into the Computer and Correcting Mistakes 29
2.3 Executing a Program 31
2.4 Saving a Program to Diskette 32
2.5 Loading a Program from Diskette 33
2.6 An Introduction to DOS 34

3. Beginning BASIC 39
3.1 Giving Names to Numbers and Words 39
3.2 Some Simple Printing Programs 43
3.3 Doing Arithmetic in BASIC 47
3.4 Displaying and Inputting Data 52
3.5 Letting Your Computer Make Decisions 60
3.6 Application 3-1: Name the State Capitals 70
3.7 More on Editing Lines 72
3.8 Some BASIC Functions 76

4. Structuring a Solution 85
4.1 A Step-wise Approach to Structuring a Solution 86
4.2 Flowcharting 89

5. More About BASIC 93
5.1 Doing Repetitive Operations 93
5.2 Application 5-1: Mortgage Payment Calculations 103
5.3 Working with Tabular Data 106
5.4 Application 5-2: The Return On An Investment 120
5.5 Adding Pizazz to Your Programs 123
5.6 Application 5-3: Investment Analysis System 127
5.7 Debugging and Testing 140
5.8 Appendix—Some Common Error Messages 146

6. Organizing a Program 149
6.1 Defining Your Own Functions 149
6.2 Subroutines 151
6.3 Some Programming Efficiencies 158

7. Using Your Computer as a File Cabinet 163
7.1 What Are Data Files? 163
7.2 Using Sequential Files 166
7.3 Application 7-1: Tracking Monthly Bills 174
7.4 Application 7-2: Attaching Subroutines to Other Programs 182
7.5 Using Random Access Files 184
7.6 Application 7-3: A Random Access Phone Directory 187
7.7 Application 7-4: An Artificial Intelligence Game 197
7.8 Appendix—ASCII Codes 202

8. A Peek into Your Apple's Memory 207
8.1 Bits, Bytes, and Where to Find Them 207
8.2 PEEKs and POKEs 208
8.3 Catching Errors Before They Cause Trouble 211

9. Playing with Words 221
9.1 Manipulating Strings 221
9.2 Application 9-1: A Word Guessing Game 226
9.3 Application 9-2: A Subprogram to Format Your Output 228
9.4 What is Word Processing? 230
9.5 Form Letters 231
9.6 Application 9-3: A Do-It-Yourself Word Processor 235

10. Playing with Pictures 239
10.1 Low Resolution Graphics 239
10.2 Application 10-1: Tic Tac Toe 246
10.3 Application 10-2: Drawing Bar Charts 252
10.4 High Resolution Graphics 255
10.5 Application 10-3: Drawing Line Graphs 260
10.6 Forming and Saving Shapes 262
10.7 Animation Principles 272
10.8 Application 10-4: Rowing Across the Screen 276

11. Playing with Sounds 279
11.1 Making Sounds in BASIC 279
11.2 Using Machine Language to Produce More Sounds 280
11.3 Application 11-1: Row, Row, Row Your Boat 283
11.4 Application 11-2: A Medley of Songs 284

12. Software You Can Buy 289
12.1 Spread Sheets—Visicalc* et al 290
12.2 Word Processing 297
12.3 Data Base Packages 299

---

*Visicalc is a registered trademark of VISICORP, Inc.

12.4 Accounting 300
12.5 Graphics 301
12.6 Educational Packages 301

13. Expanding the System 303
13.1 Communicating with Other Computers 303
13.2 Printers 304
13.3 Other Languages and Operating Systems 306
13.4 Other Extras 307

Appendix — Using Cassette Tape for Secondary Storage 308

Answers to Selected Exercises 310

Index 323

PREFACE

This book is designed to teach the computer novice how to use the Apple II Plus or Apple IIe computer. The development of the personal computer is one of the most exciting breakthroughs of our time. Indeed, the inexpensive personal computer promises to bring the computer revolution to tens of millions of people and promises to alter the way we think, learn, work, and play. This book is designed to serve as an introduction to this revolution. Accordingly, the book has two purposes. First, it instructs the reader in the operation of the Apple II Plus and the Apple IIe computer, and secondly, it illustrates some of the many ways it can be used.

We will guide the reader from the moment he or she turns on the computer for the first time, and discuss the rudiments of programming in the BASIC language. Many of the sections conclude with exercises that allow the reader to try the techniques learned. Located in the text of each section are boxed questions labeled "Test Your Understanding." The questions test you on concepts introduced in each section and comprise a built-in study guide. The answers to the "Test Your Understanding" questions are found at the end of each section.

We have covered more topics than those normally found in most other books on elementary programming. These include introductions to drawing pictures and making music with the computer. This inclusion will make this book a valuable reference even after the reader has developed some knowledge of programming.

Also included is a discussion of how to structure a problem solution. This will help the reader develop programs to solve more complex problems on the computer.

We have included several longer application programs. This is due to our conviction that the beginner should gain, in addition to learning BASIC programming, an overview of as many real-life applications as possible. These applications are designed to whet the reader's interest and may be used as preludes to further study.

Most enthusiastic personal computer users quickly upgrade their computer systems to include various equipment and other packaged programs. We have included here an introduction to Apple DOS and a discussion of other types of equipment and packaged programs available, as well as some features to be aware of when purchasing these items.

Our sincere thanks to our reviewers, Lowell Kasden, David C. Molz, Robert Roeloffs, James Rowland, Don Nielsen, Edward Nealy, Bennett L. Kramer, Thomas R. Humphrey, and John C. Totten for their careful scrutiny of the manuscript and their many helpful suggestions. Thanks also to Tony Melendez, Manager of Production, and Michael Rogers, production editor and designer, for the professional manner in which they managed the editing and production of this book; to Jessie Katz for coordinating traffic between authors, publisher, and reviewers; to Don Sellers for his cover design and original renderings of artwork in this book; and to Joan Caldwell and her sales staff for their contributions in helping this

book reach widespread distribution. Finally we would like to thank Harry Gaines, President of the Brady Company, and David Culverwell, Editor-in-Chief, for their support in this effort. We are continually impressed with their foresight in the rapidly changing and expanding world of computers.

Limits of Liability and Disclaimer of Warranty

The author and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Trademarks of material in this text

Apple II Plus, Apple IIe, and Applesoft are trademarks of Apple Computer Corporation.

Note to Authors

Do you have a manuscript or a software program related to personal computers? Do you have an idea for developing such a project? If so, we would like to hear from you. The Brady Co. produces a complete range of books and applications software for the personal computer market. We invite you to write to David Culverwell, Editor-in-Chief, Robert J. Brady Co., Bowie, Maryland 20715.

1
A First Look At Computers

1.1 Introduction

The computer age is barely thirty years old, but it has already had a profound effect on our lives. Indeed, computers are now standard equipment in the office, the factory, and even the supermarket. In the last three or four years, the computer has even become commonplace in the home as people have purchased millions of computer games and hundreds of thousands of personal computers. Computers are so common today it is hard to imagine even a single day in which a computer will not somehow affect us.

In spite of the explosion of computer use in our society, most people know very little about them. They view a computer as an "electronic brain" and do not know how a computer works, how it may be used, or how it may greatly simplify various everyday tasks. This does not reflect a lack of interest. Most people realize that computers are here to stay and are interested in finding out how to use them. If you are so inclined, this book is for you!

This book is an introduction to personal computing for the novice. You may be a student, teacher, homemaker, business person, or just a curious individual. We assume that you have had little or no previous exposure to computers and want to learn the fundamentals. We will guide you as you turn on your computer for the first time (there's really nothing to it!). From there, we will lead you through the fundamentals of talking with your computer in the Applesoft BASIC language. Throughout, we will provide exercises for you to test your understanding of the material. We will show you the many ways *you can use your computer.*

The exercises and examples will suggest programs you can write. Some of these will be designed to give you insight into how computers are used in business and industry. Others will suggest computer applications for your home. Some will be just for fun!

In addition to the exercises and examples used to illustrate specific programming concepts, several lengthier application programs are provided. These are discussed in detail to show you how the individual techniques which you are learning can be used in combination to solve more ambitious problems.

Some of the chapters in the book deal with applications which the novice may wish to skip the first time through. These include sections on some advanced applications and on making your programs a little more "friendly" (sections 5.4 through 5.6), and chapters on producing pictures and making music with your computer (chapters ten and eleven).

In addition to introducing you to the elements of programming your computer, we will also introduce some of the terminology and "jargon" which you may encounter in reading other books on the subject. Finally, we will talk a little bit about some of the other things which you can add to your computer as you become more proficient in its use.

What Is Personal Computing?

In the early days of computing (the 1940's and 1950's), the typical computer was a huge mass of electronic parts which occupied several rooms. In those days, it was often necessary to reinforce the floor of a computer room and to install special air conditioning so the computer could function properly. Moreover, an early computer was likely to cost several million dollars. Over the years, the cost of computers has decreased dramatically and, thanks to microminiaturization, their size has shrunk even faster than their price.

The first "personal" computers appeared on the market a few years ago. These computers were reasonably inexpensive, and were designed to allow the average person to learn about the computer and how to use it to solve everyday problems. These personal computers were incredibly popular and several hundred thousand of them were sold in three years.

The personal computer is not a toy, though it can be used as one. It is a genuine computer which has most of the features of its big brothers, the so-called "main-frame" computers, which still cost several hundred thousand dollars and up. A personal computer can be equipped with enough capacity to handle the accounting and inventory control tasks of most small businesses. It can also perform computations for engineers and scientists, and can even be used to keep track of home finances and personal clerical chores. It would be quite impossible to give anything like a comprehensive list of the applications of personal computers. However, the following list suggests the range of possibilities:

For the business person
- Accounting
- Record keeping
- Clerical chores
- Inventory management
- Cash management
- Payroll
- Graph and chart preparation

For the home
- Record keeping
- Budget management

Investment analysis
Correspondence
Energy conservation
Home security
Tax calculations

For the student
Computer literacy
Term paper preparation
Analysis of experiments
Preparation of graphs and charts
Project schedules
Storage and organization of notes

For the professional
Billing
Analysis of data
Report generation
Correspondence

For recreation
Computer games
Computer graphics
Computer art
Computer music

As you can see, the list is quite comprehensive. We will look at some of these applications in the examples and application programs which follow. However, if your interests are not listed, don't worry! There's plenty of room for those of you who are just plain curious about computers and wish to learn about them as a hobby.

The Apple II Plus/IIe Computer*

This book will introduce you to personal computing using the Apple Computer. This machine is an incredibly sophisticated device which uses many of the features of its main-frame big brothers. We will specifically discuss two models of the Apple Computer: the Apple II Plus and the Apple IIe. These models are very similar in operation and can use most of the same programs. We will also discuss the differences between the models. For simplicity, we will simply refer to the Apple II Computer when we mean either of the two models and will distinguish between them only when discussing the differences.

Before we begin discussing the particular features of the Apple II Computer, let's discuss some features found in all computers.

---

*Apple II Plus and Apple IIe are trademarks of Apple Computer Corporation.

1.2 What is a Computer?

A computer is a machine which can process the instructions you give it using the data you provide. It can keep track of the results of that processing and can return the results to you.

In order to explore this definition a little more thoroughly, let's break it up into component parts (breaking things up into component parts to help in understanding is something we will be doing often as we develop solutions for more and more complex problems). See Figure 1-1 for a pictorial representation of the component parts.

First of all, we say that the computer can **process data**. This includes a variety of different operations which can be very important in a lot of applications. Processing data includes the arithmetic operations which are probably among the first capabilities ascribed to the computer.

An equally important processing function, however, is the ability to make comparisons, e.g., to see whether the payment required with one way of financing is larger than with another way, whether a balance is negative, or whether someone has answered a question with a "yes" or a "no." As you shall see, the vast majority of the applications in which we are interested require some kind of logical comparison.

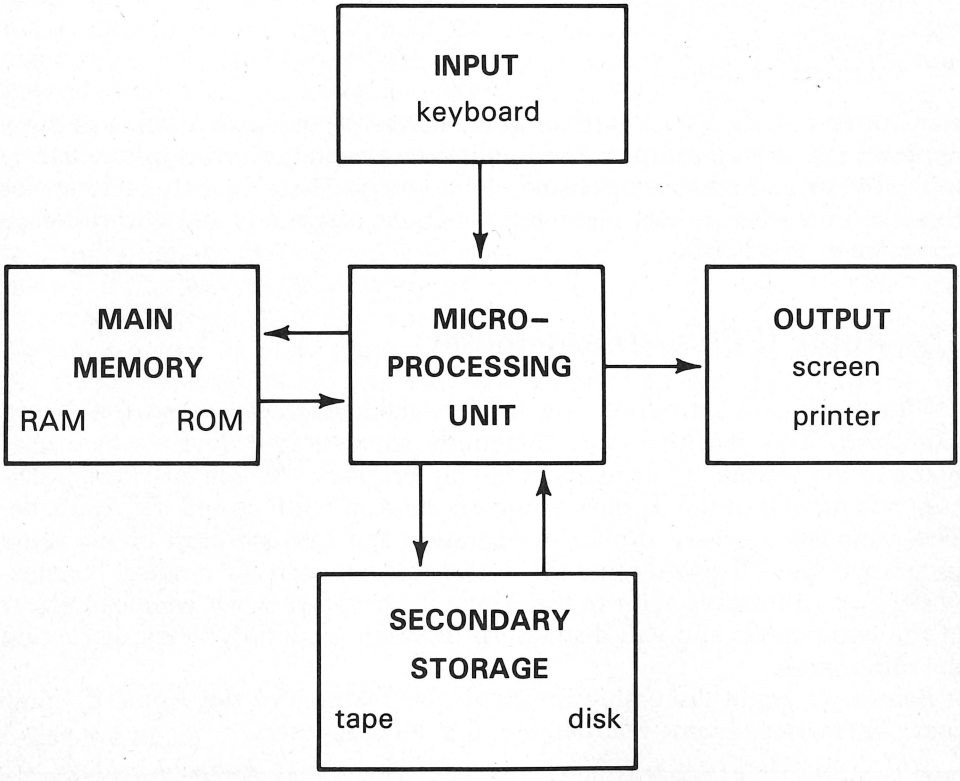


Figure 1-1. The main components of a microcomputer.

Another very important processing operation is the manipulation of characters, i.e., letters and special symbols, and numbers which you don't wish to include in arithmetic operations. For example, keeping a list of addresses and phone numbers requires being able to use the letters of the alphabet, numbers (for street addresses, zip codes, and phone numbers), and special characters like the hyphen or period.

Other processing operations allow the computer to draw pictures. These range from drawing charts and graphs to aid in analysis to the animation techniques which you may have seen in computer games. In addition, the computer can process data to create sounds and simple beeps, as well as music, and even voice.

All of the processing the personal computer does is carried out in a part of the computer called the **microprocessing unit** (or MPU). For this reason personal computers are also referred to as microcomputers. In an Apple II Computer the MPU is contained in a tiny electronic chip called a **6502 microprocessor**. The ability to put this powerful electronic circuitry on one tiny thumbnail-sized chip is one of the wonders of modern technology. Fortunately, we do not have to know the details of how this is done in order to use the computer to our advantage.

The computer can process data according to instructions which you provide. A fancier way to say this is that the computer can **control the sequence of operations**. You can give the computer a set of instructions to do several different processing operations. Once you tell the computer to "GO" (actually you say RUN), it will follow those instructions one after the other without any further intervention from you unless one of the instructions is to come back to you. Even though the computer is in control once you tell it to RUN, it is the set of instructions you have provided which is really controlling the computer. Ultimately, then, the computer is under your control.

The instructions you provide have to be written in such a way that the computer can "understand" what you want done. This requires writing the instructions according to certain rules of a computer language. A set of instructions written in a language which the computer can understand is called a **program**. Programs are frequently referred to as "software" to distinguish them from the machinery and equipment which is called "hardware."

The primary thrust of this book is to help you learn how to develop programs to get the computer to do what you want it to do, that is, to develop your own software. You will be learning how to write these programs in a computer language called BASIC (more about this later).

As the computer processes your instructions, it is able to **store intermediate results**. In fact, it must do more than that. It must keep track of the instructions you have provided as well as any and all data you wish to use. The computer stores this information in its primary storage, commonly called main "memory." Any instruction or piece of data which the computer needs must be placed in main memory before the MPU can access it. There are two types of main memory or primary storage in a microcomputer. These are called ROM and RAM.

ROM stands for "read only memory." This type of memory can be read by the computer (that is, the MPU), but you cannot record anything in it. ROM is usually reserved for the computer language which the MPU utilizes. For now, just remember that ROM contains the information necessary for the computer to understand your commands. This information is pre-recorded in the factory and is permanently situated in ROM.

RAM stands for "random access memory." This is the memory on which you can write. If you type characters on the keyboard, they are stored in RAM. Similarly, the results of calculations are kept in RAM awaiting output to you. There is an extremely important feature of RAM which you should remember:

> **If the computer is turned off, RAM is erased.**

The computer must have some means to communicate with the user, both to obtain data and to provide the results of processing. In other words, it must be able to **handle input/output operations** (or simply I/O). The input device of most computers is the computer keyboard. We will discuss the special features of the Apple II Plus and Apple IIe keyboards in section 1.3. For now, think of the keyboard as a typewriter. By typing symbols on the keyboard, you are inputting them to the computer.

The computer has a number of output devices. The most basic is the "TV screen" or **video display.** In many home installations of personal computers, this video display is the television screen. In business installations it is more likely to be a video monitor. A monitor does not have a television tuner so it cannot pick up television signals; it can only accept the signal from the microcomputer and display it via the screen. You may also use a printer to provide output on paper. In computer jargon, paper output is called **hard copy.**

While this covers all the points listed in the definition with which we began this discussion, there is one more element necessary to the practical use of a computer: the computer must **provide for long-term or secondary storage.**

As we mentioned, data which is placed in RAM is lost when the power is turned off. While we could re-enter the data and programs via the keyboard each time we wanted to use them, this would quickly become very burdensome. To make permanent copies of the programs and data, we may use either the cassette recorder or the diskette file.

The cassette recorder is simply a tape recorder which allows recording of information in a form that the computer can understand. The recording tape is the same sort you use for musical recordings.

A diskette drive records information on flexible diskettes which resemble phonograph records. Diskettes, often called "floppy disks," come in different sizes for different machines. A typical 5 1/4" diskette can store over a hundred thousand characters each! (A single-spaced typed page contains about 3,000 characters.) On the average, a diskette file can provide access to infor-

mation in much less time than a cassette file. On the other hand, diskette drives are more costly than cassette recorders.

In summary, a computer is a machine which can:

- process data, that is, perform arithmetic operations, make comparisons, and manipulate characters;
- control the sequence of operations via software;
- store intermediate results in main memory or primary storage (RAM);
- handle input/output operations, retrieve input from the keyboard and output data to the screen or printer;
- provide for long-term or secondary storage using either cassette tape or diskettes.

In the next section you will be introduced to the Apple II Plus and Apple IIe Computer keyboards and will see how the characters which you type in are displayed on the screen.

1.3 Using Your Apple II Plus/IIe Computer Keyboard

The best way to quickly master the operation of your computer is to read this book while sitting at the computer and to verify the various statements as they come up. So why not have a seat in front of your Apple II Computer? If your computer is not conveniently available, we have provided some figures which you can refer to.

Figure 1-2A is a picture of the Apple II Plus Computer; Figure 1-2B pictures the Apple IIe. As you can see, the outward appearances of these two systems are very similar. The main difference is in the keyboard design (See Figures 1-3A and 1-3B).

Let's begin by examining the keyboards. Note that they are similar to a typewriter keyboard with a few important differences. Many typewriters use the same key for the number one and a lower case letter l(el). On the computer keyboard there are separate keys for these two symbols. Similarly, it is very easy to confuse the capital letter O ("oh") and the number 0 (zero). For this reason, a computer specialist usually writes zero with a slash through it (Ø). To prevent possible confusion you should do likewise.

Note that the computer keyboard has a number of specialized keys which are not on a standard typewriter keyboard. We will discuss these keys one at a time, but first let's turn the computer on.

Turn on the power to the video display and then turn on the power to the computer itself. (The power switch is a toggle switch located in the left rear of the machine. See Figure 1-5B or 1-6B for a rear view of your computer.) Then hold down the key marked **CONTROL** on the Apple IIe or **CTRL** on the Apple II Plus and push the **RESET** key. The CONTROL key is located on the left of the keyboard; the RESET key is on the upper right. Some early models require only that you push the **RESET** key. (This was changed in later models to prevent accidentally hitting the RESET button which could

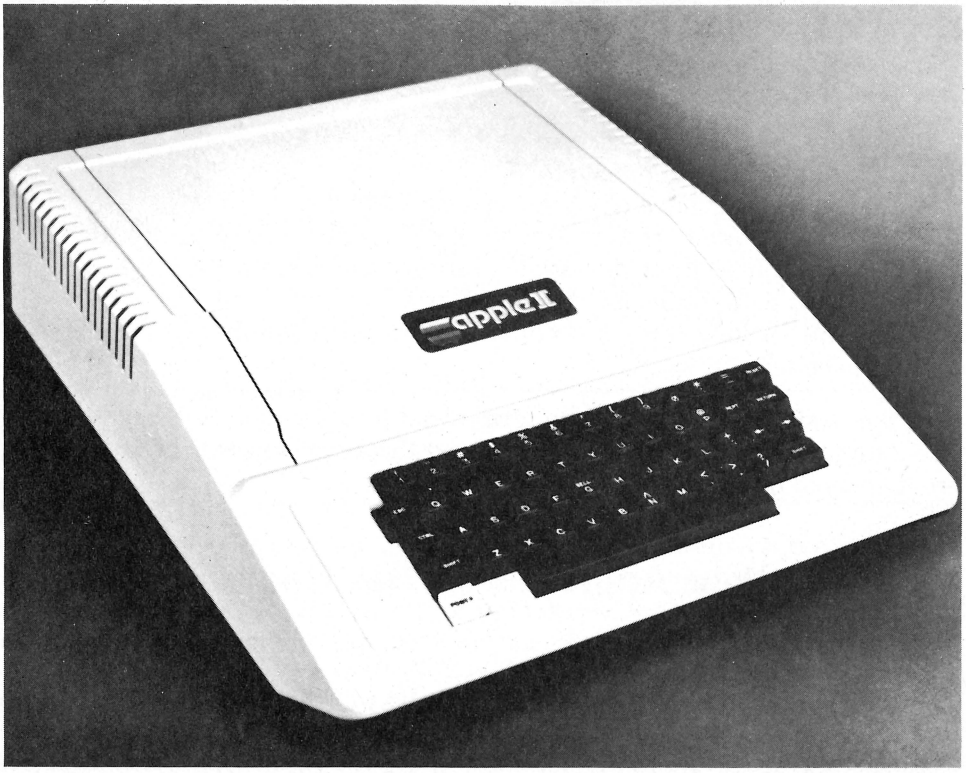


Figure 1-2A. The Apple II Plus Computer.

destroy the work in progress.) The computer will display a copyright notice and two symbols: a square right bracket (]) called the **Applesoft BASIC prompt**, and a flashing white box called the cursor. The Applesoft BASIC prompt indicates that the computer language (called Applesoft BASIC) is ready to accept your instructions.

Strike a few keys to get the feel of the keyboard. Note that as you type, the corresponding characters appear on the screen. Note also how the cursor travels along the typing line. It always sits at the location where the next typed character will appear.

Notice that some of the keys have two symbols on them: for example, the key with the number 4 on it, located in the top row of the keyboard, also has a dollar sign above it. If you strike one of these keys, the symbol on the bottom of the key appears on the screen. If you examine your keyboard, you will see that there are two keys marked **SHIFT**, one located at the bottom left of the keyboard and the other at the bottom right. By holding down one of the **SHIFT** keys and then striking one of the keys with two symbols, you can type the character at the top of the key. The long bar at the bottom of the keyboard is the space bar that can be used to type blank spaces between characters.

As you type, you should notice the similarities between the Apple II Computer keyboard and that of a typewriter. However, you should also note the differences. At the end of a typewriter line you return the carriage, either

Figure 1-2B. The Apple IIe Computer.

manually or, on an electric typewriter, with a carriage return key. Of course, your screen has no carriage to return. If you continue to type past the end of a line on your computer, the computer will automatically continue on to the next line. This is called *wraparound* and is typical of a computer's operation. It allows you to enter lines which are actually longer than the 40 characters on your screen.

When you actually finish typing a line, you must tell the computer that you are ready to move on to the next line. This is accomplished by hitting the **RETURN** key located on the right side of the keyboard. When you press the **RETURN** key, the cursor will return to the next line and position itself at the extreme left side of the screen.

The **RETURN** key has another function. **RETURN** signals the computer to accept the line just typed. Until you hit the **RETURN** key, as far as the computer is concerned, the line you just typed does not exist!

Keep typing until you are at the bottom of the screen. When you hit **RETURN**, the entire contents of the screen will move up by one line and the line at the top of the screen will disappear. This movement of lines on and off the screen is called *scrolling*.

As you may have already noticed, the computer will respond to some of your typed lines with error messages. Don't worry about these now. The computer has been built to respond only to certain typed commands. If it encounters a command it doesn't recognize, it will announce this fact

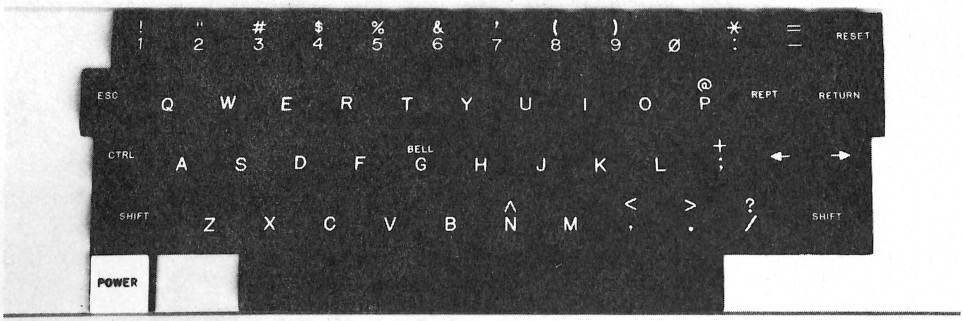


Figure 1-3A. The Apple II Plus keyboard.

Figure 1-3B. The Apple IIe keyboard.

with an error message. It is extremely important for you to realize that these errors will not harm the computer. In fact, there is little you can do to damage your computer (except by means of physical abuse, of course). Don't be intimidated by the occasional slaps on the wrist handed out by your computer. Whatever happens, don't let these "slaps" stop you from experimenting. The worst that can happen is that you might have to turn your computer off and start all over!

By this time your screen should look pretty cluttered. To clear it, type **HOME** and then press **RETURN**. All characters on the screen will be erased and only the cursor will remain. The cursor is positioned in the upper left corner of the screen, its so-called "home" position.

TEST YOUR UNDERSTANDING 1 (answers on page 13)

a. Type your name on the screen.
b. Erase the screen.

Unless you are a superb typist (most of us are at the other extreme!), you will eventually make typing errors. Let's discover how to correct them.

Type a few characters, but don't hit the **RETURN** key. Now find the key that has a left pointing arrow on it (←). This key is located on the right side of the keyboard on the Apple II Plus and on the bottom right on the Apple

IIe. It is called the backspace key. Strike this key and note that this causes the cursor to backspace one space at a time. Although the characters to the right of the cursor remain on the screen, the computer will completely ignore them. When you hit **RETURN**, these characters will disappear. Try hitting **RETURN** and see what happens.

Using the backspace key allows you to "erase" the characters it passes over, and you can retype the line correctly. This is another difference between a typewriter and a computer keyboard. Note, however, that you may use the backspace to correct lines only if they have not been sent to the computer via the **RETURN** key. We will discuss other ways to correct typing errors later in the book.

Immediately to the right of the backspace key, you will see another key with a right pointing arrow (→). This key can be used to move the cursor to the right. By using these two arrow keys you can backspace to correct a simple misspelling and then move the cursor to the right again over the letters which were typed in correctly.

TEST YOUR UNDERSTANDING 2 (answer on page 13)

Type in the word PLINT, then, using the two arrow keys, correct the word to spell PRINT.

On the left of the Apple IIe keyboard is a key marked **CONTROL**. On the Apple II Plus it is marked **CTRL** which is short for control so we will always call it the **CONTROL** key. You may already have had to use this key to obtain the Applesoft BASIC prompt as described earlier. The **CONTROL** key is always used with some other key and gives additional meaning to that key.

For example, if you have an Apple II Plus, look at the "G" key; you will see the word BELL above it (You will not have this word printed on the Apple IIe, but you still have the capability.). You cannot activate the bell using the **SHIFT** key because the bell is a non-printing character, that is, there is nothing to display on the screen. However, if you press the **CONTROL** key and then press "G" you will hear the bell (actually a high beep).

Another important function for the **CONTROL** key is that, when used in conjunction with "X," you can wipe out anything which was previously typed on a line, provided that the **RETURN** key had not been pressed. For example, if you were typing a line and had made so many mistakes that it was easier to start over rather than use the arrow keys to correct the errors, you could press **CONTROL** and "X" simultaneously. The computer would display a backward slash at the end of that line \, and the cursor would move to the next line. Try this by typing something into the computer and then hitting the **CONTROL—"X"** sequence before hitting **RETURN**. There are other uses for the **CONTROL** key which we will discover as we proceed through the text.

A similar key is the **ESC** or escape key also located on the left of the keyboard. This key is used along with some other key, though not necessar-

ily simultaneously. For example, pressing the **ESC** key, releasing it, and then pressing the "I" key will cause the cursor to move up one line each time the "I" key is pressed. We will see how this is used a little later.

Special Features of the Apple II Plus Keyboard

On the upper right of the keyboard, you will see a key marked **REPT**. This is the repeat key. If you hold down a key on the keyboard and then hold down the repeat key, the letter, number, or space indicated by that key will be repeated on the screen. Try this by holding down the "A" key and then holding the **REPT** key.

Special Features of the Apple IIe Keyboard

In addition to upper case letters, the Apple IIe keyboard is capable of generating lower case letters. This works much the same as an ordinary typewriter. If the CAPS LOCK key on the lower right of the keyboard is depressed, everything you type will be in upper case. If it is not depressed, then you will type in lower case. Try typing something in without touching the CAPS LOCK key. Now hit the CAPS LOCK key and type in the same thing. You will see that the letters shift. When we type in sets of instructions or programs in the BASIC language that we will be learning in this book, it will be necessary for you to use upper case letters exclusively.

When you are in lower case mode, you can type an individual character in capitals by holding down the SHIFT key. This is also necessary to access the special characters which are printed at the top of some keys. In this case, the SHIFT key works the same for both upper and lower case modes.

The Apple IIe is also equipped with an automatic key repeat feature. This means that if you hold a key down for more than a second, it will begin to repeat itself. Try this by holding down the "A" key. You will see that the letter "A" repeats until you remove your finger from the key.

To the left and right of the space bar (the long bar at the bottom), there are two keys with pictures of apples on them. The key to the left is called the *open Apple* function key. The key to the right is the *solid Apple* function key. These function keys are usually used in conjunction with other keys much the same as the CONTROL (CTRL on the Apple II Plus) or ESC keys. We will defer a discussion of their use until a little later.

Immediately to the right of the left (←) and right (→) arrows are two more arrow keys: a down arrow (↓) and an up arrow (↑). These keys are also cursor movement keys. If you press the down arrow, you will see the cursor move down the screen one line each time. The up arrow does not function except through program control. We will talk about how to use this key in programs a little later in the book.

The **TAB** and **DELETE** keys are also special Apple IIe keys which function only under program control. You may find these used in some of the programs which you may have purchased.

The Apple IIe keyboard also features many special symbols. For example, on the right of the keyboard are a couple of special enclosure symbols ([ ] and { }). At the bottom left is a backward apostrophe (`). You will not have occasion to use these symbols in writing BASIC programs, but they are used in some application programs. You should examine the keyboard closely to familiarize yourself with the location of these keys.

Exercises

Type the following expressions on the screen. After each numbered exercise, clear the screen.

1. 10 HELLO! I'M YOUR NEW OWNER
2. 10 ARITHMETIC
3. 10 PRINT 3+7
4. 20 LET A=3−5
5. 20 5% of 68
6. 10 IF 38 > −5
7. 10 X=5: PRINT X
8. 20 IF X>0 THEN 50
9. 10 LET X=10
   20 LET Y=50.35
10. 200 Y=X*2 − 5
    300 PRINT Y,'Y'

ANSWERS TO TEST YOUR UNDERSTANDINGS 1 and 2

1: a. Type your name, ending the line with RETURN.
   b. Type HOME followed by RETURN.
2: Type in the word PLINT but *do not* hit RETURN. Use the left arrow (←) to backspace until the cursor is over the letter L. Type in the correct letter, R, then use the right arrow (→) to move the cursor to the space following the letter T and hit RETURN.

1.4 Inside Your Apple

You have now had an opportunity to see how your computer keyboard works and how this affects the display on the screen. To gain a little better understanding of some of the terms we introduced earlier and to see how they apply to the Apple II Computer, let's take a look inside the computer. (Remember that we are using the term Apple II to refer to either the Apple II Plus or the Apple IIe.)

First, turn off the power to your computer and to the video display. If your video display unit (television or monitor) is sitting on top of the computer carefully lift it off and put it to the side. It is not necessary to disconnect the cable which leads from the video display unit to the computer, but be careful not to accidentally yank this cable. Later in this section we will look more closely at this connection.

Once again, be sure the power is turned off.

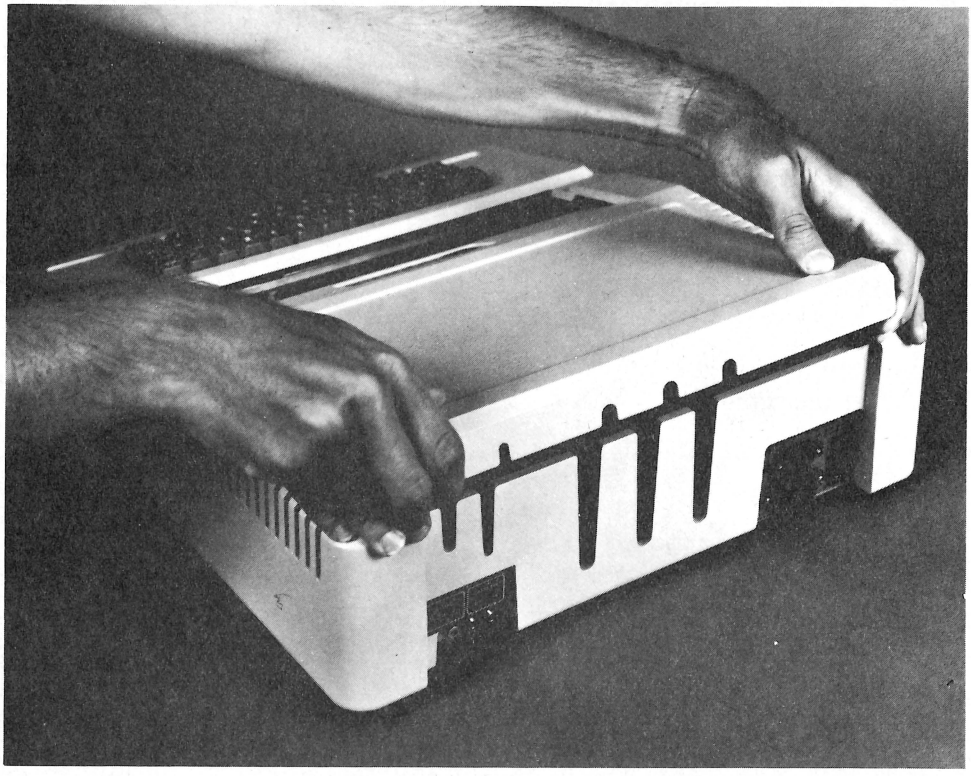


Figure 1-4. Removing the lid of the computer.

> **NEVER remove the lid of your computer unless the power is turned off.**

Carefully grasp the top of the computer lid at the rear and lift it up (see Figure 1-4). This will require a little bit of pressure, but don't overdo it. The lid is held in place by two plastic grips at either side. When the lid is released, lift it up and back, then put it in a safe place.

We will now look at the innards of the computer. Because the Apple II Plus and Apple IIe are very different inside, there is a separate section for each. You may wish to skip the one which does not apply to your machine.

Inside Apple II Plus

Now that you have the lid off of your Apple II Plus, you should see the inside, pictured in Figure 1-5A. To the left on the inside of the machine is a large, gold-colored box. This is the power pack which supplies power to your computer.

Most of what you see are small, rectangular, black boxes, arranged on a board on the floor of the computer. This main board is called the

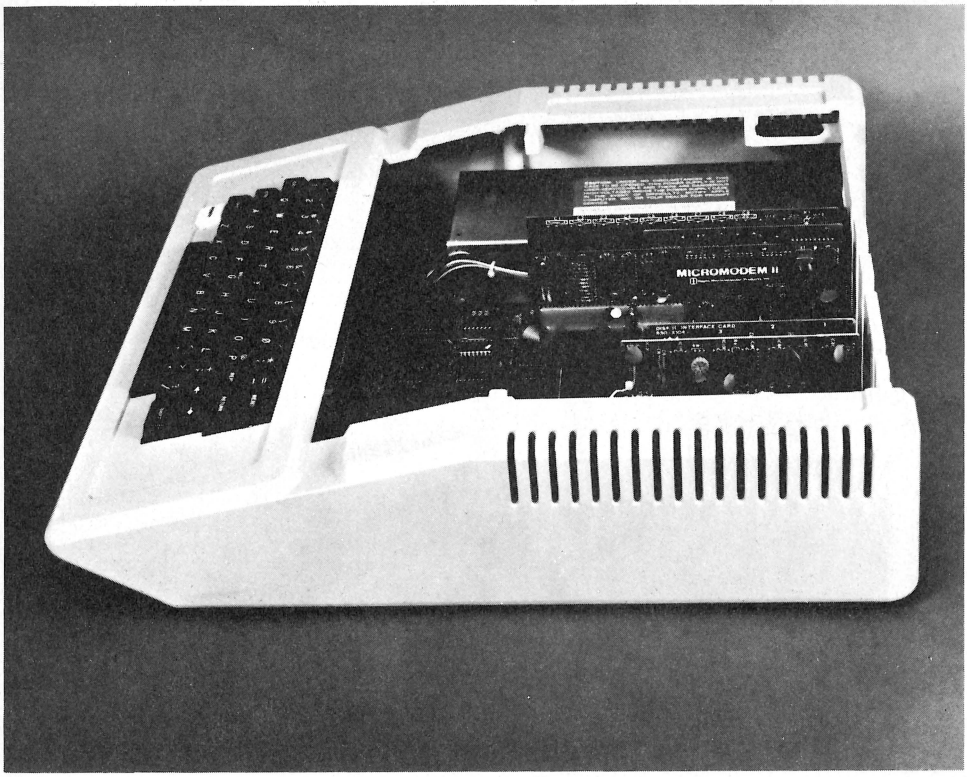


Figure 1-5A. Inside the Apple II Plus.

"motherboard." It serves to connect all the individual pieces of the computer logic together. The rectangular black boxes are called integrated circuits (sometimes large scale integrated circuits or LSI circuits) or "chips." They actually contain all the circuitry which allows the computer to perform processing operations and store data.

Toward the center of the motherboard is a chip which is somewhat larger than the others. This is the microprocessing unit or MPU. In the Apple II Plus, this is a 6502 chip. The number refers to the design of the chip. Different microcomputers use differently designed microprocessor chips. This is one reason that software developed for the Apple II Plus will not work on other microcomputers. Other popular chips in use are the Z80 used in the Radio Shack (Tandy) TRS-80, and the 8088 used in the IBM PC.

The computer's "memory" is also contained on chips. These are the other rectangular black boxes in the computer. The **ROM** (read only memory) chips are located immediately below the 6502 MPU. The first of these is the "Autostart ROM" chip which we will discuss later. It allows the computer to automatically communicate with the disk drive when the computer is turned on.

RAM (random access memory) chips are located below the ROM chips. This memory is where things typed into the keyboard are stored in the computer. It is volatile; that is, the contents of RAM are erased when power is turned off.

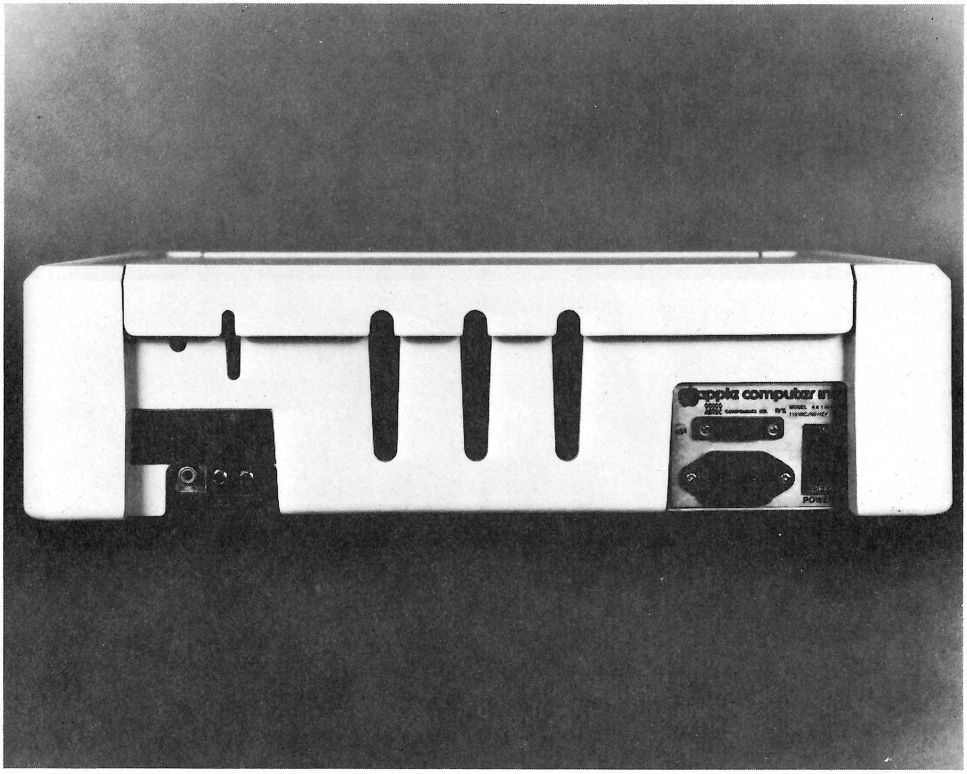


Figure 1-5B. A rear view of the Apple II Plus.

There are other chips in the computer also. These control things like the video display unit.

To the rear of the Apple II Plus, in back of the MPU, are eight slots which are used to connect extra equipment to the computer. The eight slots are numbered from 0 at the extreme left to 7 on the right.

If your disk drive is connected, you will have a board in one of these slots with a flat cable leading from the board to the disk drive. The board in the slot allows the computer to control the operations of the disk. Usually the "disk drive controller board" will be in slot number 6. If you have a printer attached, there will be another board in one of the slots (usually slot 1) connecting the computer and the printer.

These boards are called "interface" boards or cards because they allow your Apple II to interface with some external device. These external devices may be a disk drive, a printer, a music synthesizer, or a device to allow a telephone link, as well as many other special pieces of equipment. These external devices are called "peripherals." We will discuss some possible peripherals later in the book.

Slot 0 is reserved for special boards which allow you to add additional RAM or use other computer languages besides the BASIC language we will cover in this book.

If you have purchased some games to use with your Apple II Plus, you may also have "paddles" or a "joystick" attached. These are controls which

allow you to move things about on the screen without typing on the keyboard. Such devices are connected via the I/O (input/output) port located to the right of the slots. If you look closely, you will see the word GAME printed on the motherboard next to the I/O port.

Under the keyboard itself, on the left, is the Apple II Plus speaker. We will learn how to use this speaker in programs discussed in Chapter Eleven.

A rear view of the Apple II Plus is provided in Figure 1-5B. On the right of the picture (left rear of the computer) you will see the power switch for the Apple II Plus. Immediately next to it is the connection for the power cord. You plug the cord into a standard 110 volt outlet.

There are several openings at the rear of the computer. These allow the cables leading from your disk drive controller board or printer interface board inside the computer to attach to those external devices if you have them.

You also have connections leading to your video display unit. If you have a monitor, it is connected via the video output jack. This jack is located at the right rear of the computer and extends out the back. In Figure 1-5B you can see the video output jack on the extreme left (extreme right on the back of the computer).

If you are using a television set for your video display, it is connected a little differently. In this case the television set is connected to the computer through an RF (radio frequency) modulator. This is a device which changes the signal from the computer into one which can be displayed on the television set.

The RF modulator consists of two parts. One must be attached to the computer and is usually located inside on the right wall. It is connected to the computer through a set of auxiliary video output pins located on the motherboard just behind the video output jack. A cable then runs from this part of the RF modulator out the back of the computer to a second part which is attached to the television set via its antenna leads. You will have an RF modulator only if you use a television set for output. It is an accessory which must be purchased separately and is not a standard feature of the Apple II Plus.

At the rear of the computer, next to the video output jack, are the cassette input and output jacks. This allows a tape cassette player/recorder to be attached to the Apple II Plus computer and used for long-term storage of programs and data. If you do not have a disk drive, you will want to use a tape cassette for storage purposes. However, the diskettes provide an easier, quicker, and more reliable means of storage.

Inside Apple IIe

Now that you have the lid off of your Apple IIe, you should see the inside, pictured in Figure 1-6A. To the left on the inside of the machine is a large gold-colored box. This is the power pack which supplies power to your computer.

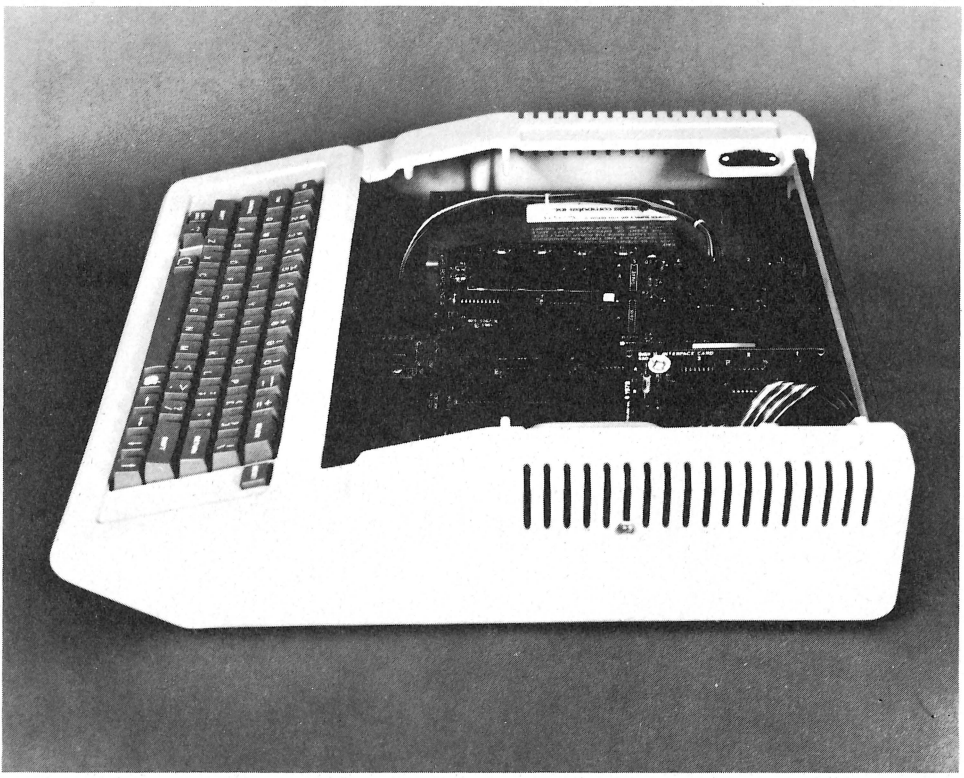


Figure 1-6A. Inside the Apple IIe.

Most of what you see are rectangular black boxes arranged on a board on the floor of the computer. This main board is called the **"motherboard."** It serves to connect all the individual pieces of the computer logic together. The rectangular black boxes are called integrated circuits (sometimes large-scale integrated circuits or LSI circuits) or "chips." They actually contain all the circuitry which allows the computer to perform processing operations and store data.

Toward the center of the motherboard are three chips which are somewhat larger than the others. The one closest to the rear of the machine is the **microprocessing unit** or **MPU**. In the Apple IIe, this is a 6502 chip. The number refers to the design of the chip. Different microcomputers use differently designed chips. This is one reason that software developed for the Apple IIe will not work on other microcomputers (software for the Apple II Plus and the Apple IIe is generally compatible because they do use the same MPU, and other design decisions were made to see that a great deal of compatibility did exist). Other popular chips in use are the Z80 used in the Radio Shack (Tandy) TRS-80, and the 8088 used in the IBM PC.

The computer's "memory" is also contained on chips. These are the other rectangular black boxes in the computer. RAM (random access memory) chips are located to the right near the keyboard. There is a white line printed on the motherboard above them. This memory is where things typed into

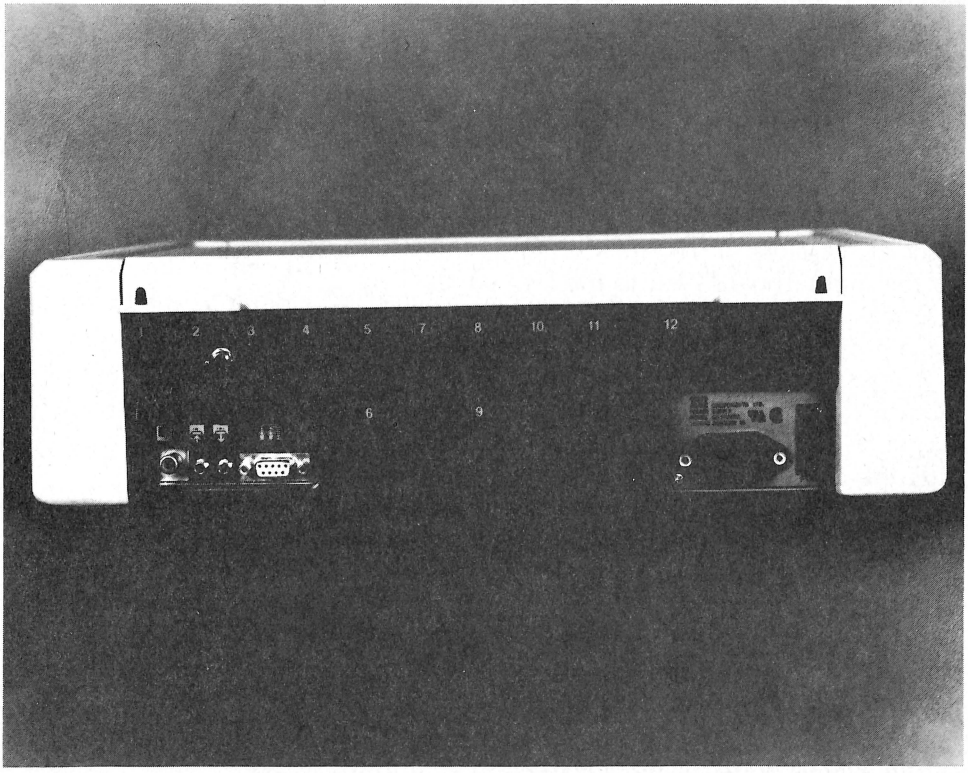


Figure 1-6B. A rear view of the Apple IIe.

the keyboard are stored in the computer. It is volatile; that is, the contents of RAM are erased when power is turned off.

Other chips are ROM (read only memory) chips. They allow the computer to do many things such as automatically communicating with the disk drive when the computer is turned on and controlling the video diplay unit and the keyboard.

To the rear of the Apple IIe, in back of the MPU, are seven slots which are used to connect extra equipment to the computer. The seven slots are numbered from 1 at the extreme left to 7 on the right. If your disk drive is connected, you will have a board in one of these slots with a flat cable leading from the board to the disk drive. The board in the slot allows the computer to control the operations of the disk. Usually, the "disk drive controller board" will be in slot number 6. If you have a printer attached, there will be another board in one of the slots (usually slot 1) connecting the computer and the printer. These boards are called "interface" boards because they allow your Apple II to interface with some external device.

These external devices may be a disk drive, a printer, a music synthesizer, or a device to allow a telephone link, as well as many other special pieces of equipment. These external devices are called "peripherals." We will discuss some possible peripherals later in the book.

There is an additional expansion slot, called slot 0, located just to the right of the power pack in the center of the motherboard. Slot 0 is reserved for

special boards which allow you to see an 80-column display instead of 40 or use other computer languages besides the BASIC language which we will cover in this book.

If you have purchased some games to use with your Apple IIe, you may also have "paddles" or a "joystick" attached. These are controls which allow you to move things about on the screen without typing on the keyboard. Such devices are connected via the I/O (input/output) port located to the right of the slots. If you look closely, you will see the word GAME printed on the motherboard next to the I/O port.

Under the keyboard itself, to the left, is the Apple IIe speaker. We will learn how to use this speaker in programs discussed in Chapter Eleven.

A rear view of the Apple IIe is provided in Figure 1-6B. On the right of the picture (left rear of the computer), you will see the power switch for the Apple IIe. Immediately next to it is the connection for the power cord. You plug the cord into a standard 110 volt outlet.

There are several openings at the rear of the computer. When you first receive your Apple IIe, these are covered with little plastic inserts. These inserts can be removed as needed to allow the cables leading from your disk drive controller board or printer interface board inside the computer to attach those external devices if you have them. Instructions for removing these plastic inserts and replacing them with inserts to hold the cables are provided in the *Apple IIe User's Manual,* Apple Product #A2L2001.

You also have connections leading to your video display unit. If you have a monitor, it is connected via the video output jack. This jack is located on the right rear of the computer and extends out the back. In Figure 1-6B, you can see the video output jack on the extreme left (extreme right on the back of the computer).

If you are using a television set for your video display, it is connected a little differently. In this case, the television set is connected to the computer through an RF (radio frequency) modulator. This is a device which changes the signal from the computer to one which can be displayed on the television set.

The RF modulator consists of two parts. One must be attached to the computer and is usually located inside on the right wall. It is connected to the computer through a set of auxiliary video output pins located on the motherboard just behind the video output jack. A cable then runs from this part of the RF modulator out the back of the computer to a second part which is attached to the television set via its antenna leads. You will have an RF modulator only if you use a television set for output. It is an accessory which must be purchased separately and is not a standard feature of the Apple IIe.

At the rear of the computer, next to the video output jack, are the cassette input and output jacks. This allows a tape cassette player/recorder to be attached to the Apple IIe computer and used for long term storage of programs and data. If you do not have a disk drive, you will want to use a tape cassette for storage purposes. However, the diskettes provide an easier, quicker, and more reliable means of storage.

A Word About RAM

In speaking about the size of a computer, people do not usually refer to the physical size of the hardware. Instead they talk about the amount of storage, usually random access memory, which the computer has.

The memory size is measured in units called **bytes**. One byte is enough memory to store one character, that is, one letter. The computer itself does not store letters directly, but stores everything as numeric codes consisting only of zeroes and ones. Each individual zero or one in a code is called a **bit** for binary digit. The codes used in microcomputers use eight bits, that is, eight zeroes and ones to code each letter. Therefore, a **byte** is eight bits long. The size of memory, therefore, is expressed as the number of kilobytes of RAM. The prefix *kilo*, as used in this context, represents 1024. Thus, a 48K machine is a computer with 48 kilobytes or 48 x 1024 bytes of storage, enough to store 49,152 characters. A 64K machine can store 64 x 1024 or 65,536 characters.

1.5 Using the Apple II Plus/IIe Disk Drive

The Apple II Computer has provisions for installing one or more disk drives. Each disk drive must be connected to a disk controller card which is inserted into one of the empty slots in the main system unit. A single controller card may control one or two disk drives.

A particular disk drive is identified by two numbers: the slot number in which its controller card is inserted and the number 1 or 2 indicating the disk drive on the particular controller card. The slot number is indicated in commands by the letter S, and the drive number is indicated by the letter D. For example,

```
S3, D2
```

refers to disk drive 2 attached to the controller card in slot 3.

The Apple II Computer allows you to build a customized system by inserting various circuit boards into the various slots of the system unit. It is customary for slot 6 to contain the disk controller board. **For the rest of this book, we will assume that the circuit boards are arranged in this fashion.**

To store information the disk drives use 5¼-inch floppy disks. Each disk can accommodate approximately 143,000 characters (about 50 single-typed pages). Figure 1-7 illustrates the essential parts of a disk. The jacket is designed to protect the disk. The interior of the jacket contains a lubricant which helps the disk rotate freely within the jacket. The disk is sealed inside. You should never attempt to open the protective jacket.

The disk drive reads and writes on the disk through the **read-write** window. Never, under any circumstances, touch the surface of the disk. Diskettes are very fragile. A small piece of dust or even oil from a fingerprint may damage the disk and render all the information on it totally useless.

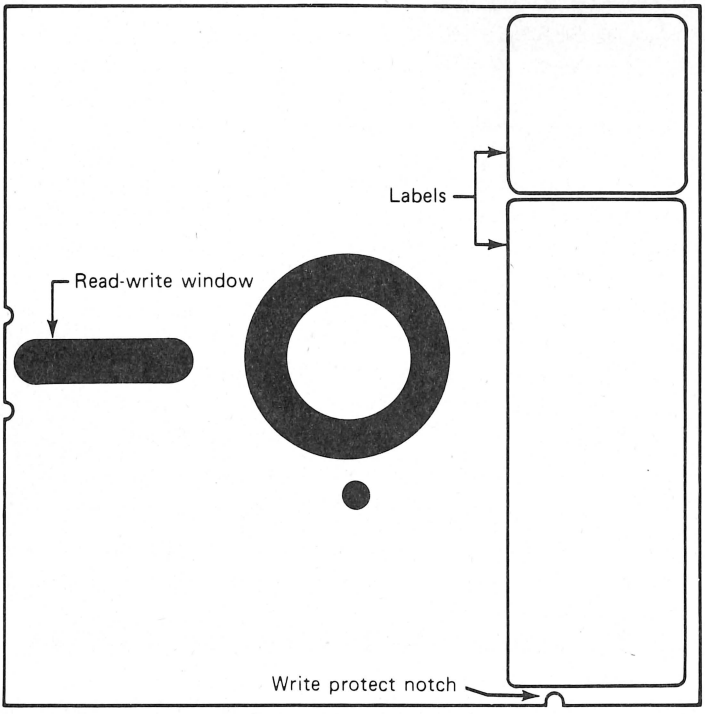


Figure 1-7. A floppy disk.

The **write protect notch** allows you to prevent changes of information on the disk. When this notch is covered with one of the metallic labels provided with the disks, the computer may read the diskette, but it will not write or change any information on the diskette. To write on a diskette, you must uncover the write protect notch.

To insert a diskette into a disk drive, open the door of the drive. Turn the diskette so that the label side is facing up and the read-write window is facing toward the rear. Push the diskette into the drive. Close the drive door.

Diskettes are extremely fragile. Here are some tips in using them:

1. Always keep a diskette in its paper envelope when not in use.
2. Store diskettes in a vertical position just as you would a phonograph record.
3. Never touch the surface of a diskette or try to wipe the surface of a diskette with a rag, handkerchief, or other piece of cloth.
4. Keep diskettes away from extreme heat, such as that produced from radiators, direct sun, or other sources of heat.
5. Never bend a diskette.
6. When writing on a diskette label, use only a felt-tipped pen. Never use any sort of instrument with a sharp point.
7. Keep diskettes away from magnetic fields, such as those generated by electrical motors, radios, televisions, tape recorders, and other devices. A strong magnetic field may erase data on a diskette.

8. Never open the drive door or remove a diskette while the drive is running. (You can tell if a drive is running by the sound of the motor and the "in use" light on the front of the drive.) Doing so may cause permanent damage to the diskette.

The above list of precautions may seem overwhelming to someone starting out. But once you have used them to construct a suitable set of procedures for handling and storing diskettes, you will find that the diskettes are a reliable, long-lasting storage medium.

Starting the Computer with the Disk Drive

As mentioned earlier, the rules for interpreting your commands are contained in ROM. In the Apple II this includes Applesoft BASIC. Also contained in ROM are instructions to allow the computer to manage the activities which go on inside and to coordinate the flow of information between the keyboard, the video display, a tape cassette, a printer, and the many other peripherals you may have connected to your computer. This set of instructions is called the **operating system**. Apple refers to this as the **"monitor."**

This operating system or monitor is not powerful enough to control the flow of information to and from the disk drives. For this purpose Apple provides you with an additional set of instructions called Apple DOS (Apple Disk Operating System—pronounced Apple-doss). This system is contained on a **System Master diskette** which is included when you purchase your disk drives.

In order to make use of your disk drives, it is necessary to read DOS into the computer so that it can supplement the operating system in ROM. This process is called "bootstrapping" or more commonly "booting the disk." To "boot" the disk for your Apple II Computer, follow this procedure:

1. Insert the System Master diskette into drive 1.
2. Close the drive door.
3. Turn on the computer. Apple II Computers come equipped with a feature called "Autostart ROM." When you turn the computer on, it automatically goes to drive 1 and "boots" the disk which is in that drive.
4. Diskette drive 1 will turn on and you will hear the whirring action of the disk drive. When the disk drive is in use, the red light on the door will be lit. Never open the door while the red light is on. Doing so may cause you to lose some or all of the data on the diskette.
5. DOS will display something like

```
DOS VERSION X.X                    DATE
APPLE II PLUS OR ROMCARD           SYSTEM MASTER
```

6. The computer will then display the Applesoft BASIC prompt. You may now use diskettes to save programs.

NOTE: You can always reboot the system without turning the computer off by typing **PR#6** where 6 is the slot number containing the disk drive controller board. This is sometimes called a "warm boot" because the computer is already on and, thus, warmed up. Turning the computer off and then allowing the disk to boot automatically when the power switch is turned on is called a "cold boot." In the Apple IIe you can also effect a "warm boot" by holding down the Open-Apple function key located to the left of the space bar and simultaneously hitting the CONTROL—RESET sequence.

Initializing a Diskette

You have used the system master diskette to boot the disk, that is, load DOS into RAM. If you examine this disk, you will see that it is write-protected; it does not have a notch which would allow you to save programs on it. Because we want to be able to save the programs we write, it is necessary to have a separate diskette on which we can write. It is usually a good idea to have extra diskettes available for this purpose.

In order to use a diskette with your disk operating system, it is necessary to initialize it, that is, put it in a form so that the operating system can store information on it. Once you have booted DOS, you can initialize extra diskettes for program and data storage. To do this, you type in a progam called the "HELLO" program.

Here are the steps you need to follow. After booting DOS, type NEW. Then type in the "HELLO" program. This can be any program at all. For now, let's use the following one-line program:

```
10 PRINT "HELLO"
```

Typing in this one-line program enters it into RAM. Now remove the system master diskette from drive 1 and insert a blank diskette. Type in

```
INIT HELLO
```

followed by a RETURN. You will see the red light on your drive go on. When it goes off again, the diskette will be initialized and can be used to boot the disk operating system and to store programs. Note that if you have two disk drives, you can also insert the blank disk into disk drive 2. In this case the command to initialize is

```
INIT HELLO,D2
```

Assumed System Configuration

This book assumes that the user has, at a minimum, either an Apple II Plus with at least 48K of RAM or an Apple IIe with 64K RAM plus a video display unit which could be a television or a monitor. It also assumes at least one disk drive with controller board attached to slot 6. All application programs provided will run on the minimum configuration.

If you have a printer for your system, you can print programs by activating the printer using **PR#n** where n is the slot number for your printer interface card (usually number 1). You may also need to send some other control signals to the printer or interface board. See the manuals which came with these to check this out. In Chapter Seven we will discuss how to turn the printer on and off within your program. Several commands we will be learning may operate differently on your printer. We will note these as we learn the commands.

2

Running Programs On Your Apple II Plus/IIe Computer

2.1 Computer Languages and Programs

In Chapter One we learned that a computer is a machine which can process data, control the sequence of operations through programs, store intermediate results, handle input/output operations, and provide for long-term storage. We also learned how to manipulate the keyboard and the screen of the Apple II Computer. (Remember that when we use the term Apple II in this book, we are referring to the Apple II Plus and the Apple IIe.) In this chapter we will demonstrate each of these features using a simple program which the authors have written.

Let's consider a problem we can use for illustration. Assume that we were interested in writing some kind of a solution procedure for converting distances from feet to meters. We look in an available reference manual, a dictionary perhaps, and we learn that

1 foot = .3048 meters

If we want to convert three feet to meters, we would multiply 3 by .3048 to obtain .9144 meters. A general set of instructions for solving this kind of problem might be:

a. get the number of feet
b. multiply the number of feet by .3048
c. write down the answer

Note that this is a general solution procedure because it will work for any number of feet. The number of feet, then, is a variable. The answer will also vary as we change the number of feet. The conversion factor, on the other hand, remains constant. We can, therefore, write the conversion factor into our set of instructions in numeric form.

The general set of instructions for converting feet into meters (a,b,c above) is written in English. It needs to be translated into a language the computer

can understand. We need a computer program to convert feet into meters. Such a program is shown in Figure 2-1.

```
10 PRINT "PROGRAM TO CONVERT FEET TO METERS"
20 INPUT "NUMBER OF FEET? ";FT
30 MT=.3048 * FT
40 PRINT "EQUIVALENT # OF METERS=";MT
50 END
```

Figure 2-1. Program to convert feet to meters.

The sample program is written in a computer language called **BASIC** (**B**eginners **A**ll **P**urpose **S**ymbolic **I**nstruction **C**ode). This is the best computer language to learn first since it is the most elementary computer language used by your Apple II Computer.

The first version of BASIC was developed by John Kemeny and Thomas Kurtz at Dartmouth College. In the next few chapters, we will concentrate on learning the fundamentals of Applesoft BASIC, the version available on your Apple II.

Many people think of a computer as an "electronic brain" which somehow has the power of thought. This is very far from the truth. The electronics of the computer and the rules of the Applesoft BASIC language allow it to recognize a very limited vocabulary and to take various actions based on the data given to it. It is very important to recognize that the computer does not have "common sense." The computer will attempt to interpret whatever data you input. If what you input is a recognizable command, the computer will perform it. It does not matter that the command makes no sense in a particular context. The computer has no way to make judgments. It can only do what you instruct it to do.

Because of the computer's inflexibility in interpreting commands, you must tell the computer exactly what you want it to do using the correct rules for constructing the statement. If you communicate a command in an incorrect form, you won't damage the machine in any way. You simply won't obtain the results you desire. If you communicate a command which the computer cannot interpret, it will respond by displaying "SYNTAX ERROR". You probably saw this message many times while you were learning to use the keyboard in Chapter One. This means you have violated the rules for statement construction.

Before we get into the detailed rules of the BASIC language, let's look briefly at the sample program and then use it to learn about some fundamental capabilities of the computer.

Examining the program in Figure 2-1, we see that the first instruction (line number 10) tells the computer to PRINT the words enclosed in quotes. This statement is rather easy to understand. We are simply telling the computer to have the sentence in quotes appear on the screen (we could also work with a printer, but we will save this discussion for later). The PRINT statement is an output instruction in BASIC.

The next statement (line number 20) is an INPUT statement. It corresponds to instruction (a) in our English language solution. It instructs the computer to accept

input data from the keyboard. Because the computer will wait for the person at the keyboard to type in a number, we also include with the input statement a prompt or cue message, that is, some short message or reminder telling the user what the computer wants. Thus, the statement begins with the word INPUT. This is followed by the prompt enclosed in quotes. Finally, the prompt message is followed by a semicolon and then the letters FT. The authors chose the letters FT to stand for feet. The value for feet may vary each time the computer program is run; thus, when we write the program, we use some symbol for the variable. When the computer reads in the number which is typed at the keyboard, it places that number in RAM and associates it with the name FT. Any time we wish to use that number, we can access it by asking for FT. We will say more about variables later, but for now, the symbol FT serves the same purpose as saying 'number of feet' in the English language solution.

The next statement in the program (line number 30) instructs the computer to process the data, in our case to calculate the number of meters. The asterisk (*) is the computer instruction for multiplication. MT is the symbol which the authors chose to represent the number of meters, which is a variable.

The next statement (line number 40) is another output statement. This time it is designed to have the computer display the answer. We instruct the computer to print out the phrase "EQUIVALENT # OF METERS=" and then to print the value of the variable MT.

The last statement in the program (line number 50) is the END statement which tells the computer that there are no more instructions you wish to execute. Notice that each statement in the BASIC program begins with a line number. These line numbers do not have to be consecutive, but the computer will automatically keep them in numerical order and execute them in that order.

TEST YOUR UNDERSTANDING 1 (answer on page 29)

Write an English language set of instructions to add any two numbers together.

ANSWER TO TEST YOUR UNDERSTANDING 1

1: a. Get two numbers to add.
b. The sum is the first number plus the second number.
c. Write down the sum.

2.2 Entering a Program Into The Computer And Correcting Mistakes

At this point, we have our BASIC program written on paper. We want to communicate these instructions to the computer so we can instruct the com-

puter to do the calculations for us. We can type the instructions into the Apple II Computer using the keyboard.

To be sure that there is no other program already in RAM, type in the command **NEW**. This tells the computer that you are about to begin on a new program and instructs it to erase any other program from main memory. Because the computer can only have one program in RAM at a time, if you were to begin typing in your program statements without saying NEW, the computer would think you were adding these instructions to any which might already be in RAM. Remember:

RAM can contain only one program at a time.

Now that we have typed in NEW, we are ready to type in the set of instructions to convert feet to meters. Type in one line at a time beginning with the line number and ending with the RETURN key. If you should make some typing mistakes, don't worry about them now. We will correct them later. Some of the statements will be too long to fit on one line. Just keep on typing until you come to the end of the statement. The computer will automatically continue on the next line if necessary. This is a feature called "wrap around."

When you have finished typing in the entire program, you can ask the computer to display the program by typing LIST. If you have an Apple IIe, you can also obtain a listing of an entire program by hitting the ESC key and then hitting L. Try this now and examine your program for any typing errors.

You already know how to correct typing mistakes if you notice them before hitting RETURN. (This is done with the backspace key (←) on the Apple II Computer keyboard.) In addition, if you wish to eliminate a line which you have typed in before hitting **RETURN**, you can erase it by hitting the **CONTROL-X** sequence (**CONTROL** and **X** keys simultaneously).

Often, however, you don't notice an error until after you have sent the instruction to the computer, that is, hit the RETURN key. The most straightforward way to correct such errors is simply to retype the line. The Apple II Computer recognizes each instruction by its line number. If you try to type in a second line with the same line number, it will merely erase the first line and replace it with the second. Let's try this with the program you have just typed in.

If you have asked the computer to **LIST** the program for you, the program should look like Figure 2-1, though it may contain some errors. To illustrate the replacement feature, let's assume that we do not like the message we have provided in line 20, that we would rather say "NUMBER OF FEET TO CONVERT?". That means we would like the new line 20 to appear as

```
20 INPUT "NUMBER OF FEET TO CONVERT? ";FT
```

Type this line into the computer, then LIST your program again. You should now see that the program contains your new line 20 in place of the original.

If you have made any mistakes in typing your program, you can correct them by simply retyping the line with the same line number.

Let's try one more thing before we actually ask the computer to follow our instructions. Type in the number 10 immediately followed by the RETURN. Now LIST your program. You should see that the statement number 10 no longer appears in the program. You can entirely delete a line from the program simply by typing in its line number immediately followed by the RETURN.

Retype line 10 and any other lines in which you made typing errors so that your program looks like Figure 2-1. The correct version of the program to convert feet to meters is now in RAM. LIST it to be sure.

2.3 Executing a Program

At this point we have examined a simple BASIC program and have entered it into RAM via the keyboard. Though we have given the instructions to the computer in a language which it can understand, the computer has not yet acted upon those instructions. Indeed, because the computer is "at your service," it will not act upon your instructions until you explicitly tell it to do so.

To instruct the computer to follow your instructions (that is, to execute your program), simply type in RUN. You should see the result of your first two instructions appear on the screen. This is shown in Figure 2-2A. The computer has printed out what you requested in line 10 and has printed the cue or prompt which you desired to accompany the request for data.

You should also see the cursor flashing after the question mark. This is the computer's way of telling you that it is waiting for some data from you. Let's respond by typing in the number 3 followed by RETURN (see Figure 2-2B). Immediately, the computer will print out the result in the format which you requested, that is, preceded by the statement "EQUIVALENT # OF METERS=."

```
PROGRAM TO CONVERT FEET TO METERS
NUMBER OF FEET?
```

Figure 2-2A. Results of the first two instructions.

```
PROGRAM TO CONVERT FEET TO METERS
NUMBER OF FEET? 3
EQUIVALENT # OF METERS=.9144
```

Figure 2-2B. The answer appears when input is provided.

```
PROGRAM TO CONVERT FEET TO METERS
NUMBER OF FEET? 5280
EQUIVALENT # OF METERS=1609.344
```

Figure 2-2C. A run with 5280 feet.

If you wish to convert some other number of feet, you can do so by reRUNning the program. In Figure 2-2C the program is run using an input value of 5280 which is the number of feet in a mile. We can see that a mile is 1609.344 meters. Run the program with other data as many times as you would like until you feel comfortable with this operation.

2.4 Saving a Program to Diskette

We have now written a program which is a general procedure for converting feet to meters. If at some time in the future we would wish to use this program again (and we will be using it many times throughout the book in order to add on new features which we learn about BASIC), we could simply type it back into RAM via the keyboard.

There is a more convenient way to re-enter a program into RAM. This method requires saving the program on tape or diskette. At a later point in time, the program can be read back into the computer from this secondary storage medium. This is called "loading" the program. In fact, if you have purchased any programs from the computer store (for example, a computer game), you did not obtain a paper copy which you then had to type in. You purchased the software (program) on a tape or diskette.

In order to save programs on diskette, it is first necessary to initialize the Disk Operating System (DOS) and to have an initialized data disk available. You may wish to review Section 1.5 before proceeding. The material to follow assumes you have some familiarity with using the disk drive. It assumes that you have booted DOS.

We must first assign the program a name, which may consist of any string of up to 30 letters or numbers. Suppose that we choose the name FEET-TO-METERS for our program. We may save this program on the diskette. If you have more than one disk drive, the program may be saved on the diskette in either disk drive. To save FEET-TO-METERS on drive 1, for example, we would use the command

```
SAVE FEET-TO-METERS,D1
```

When the computer finishes writing a copy of the program onto the designated diskette, it will display the prompt. Saving a program does not alter the copy in RAM.

TEST YOUR UNDERSTANDING 1 (answer on page 33)

This is a BASIC program to add two numbers together.

```
10 INPUT "TYPE IN THE FIRST NUMBER TO ADD ";N1
20 INPUT "TYPE IN THE SECOND NUMBER TO ADD ";N2
30 S=N1 + N2
40 PRINT "THE SUM IS ";S
50 END
```

1. Type this program into RAM via the keyboard (remember to type in NEW to clear RAM first).
2. LIST the program to check for mistakes and correct these typing errors.
3. RUN the program a few times with different data values.
4. SAVE the program on diskette (call it SUM).

ANSWER TO TEST YOUR UNDERSTANDING 1

After typing in your program and correcting the errors, type RUN. The output should look like this (this example uses input data values of 2 and 3).

```
TYPE IN THE FIRST NUMBER TO ADD 2
TYPE IN THE SECOND NUMBER TO ADD 3
THE SUM IS 5
```

To save the program on diskette type

```
SAVE SUM,D1
```

(or `SAVE SUM,D2` if using drive 2)

2.5 Loading a Program from Diskette

To see that the program which you have written is really saved for future use, let's turn the computer off and start all over. Remember that turning the computer off erases everything in RAM. This means that your program FEET-TO-METERS (or your program SUM if you tried the TEST YOUR UNDERSTANDING exercise) will be erased. Because we have saved it on a long-term storage medium we should be able to get it back into RAM without retyping. This is called LOADing the program.

Turn the computer on again and boot your disk (remember that this means to load DOS into RAM). We will read the program back from the diskette into RAM. To do this we use the **LOAD** command. For example, if the diskette containing your program is now in drive 2, we use the command

```
LOAD FEET-TO-METERS,D2
```

After you have loaded the program back into RAM, type LIST to see that the program is indeed available. You can now RUN it again. This version of your program will remain available to you by loading it from your secondary storage medium. We will discuss this later as we begin to make changes to FEET-TO-METERS.

TEST YOUR UNDERSTANDING 1

LOAD the program SUM, which you saved during the TEST YOUR UNDERSTANDING in the previous section, back into RAM from the diskette. LIST the program, then RUN it.

2.6 An Introduction to DOS

In Chapter One we discussed a little about using the disk drive to provide secondary storage. In that chapter you learned how to load the Disk Operating System into RAM and how to initialize a data disk. DOS also provides a number of additional commands which make Applesoft BASIC more flexible and easy to use.

In this chapter we learned two additional DOS commands, those required to SAVE and LOAD files. It is beyond the scope of this book to cover all the special features of DOS. For a complete description of these features, we refer you to **Apple II—The DOS Manual**. However, here are a few of the most useful features of DOS.

Making Additional Copies of Diskettes

Good programming practice dictates that you keep duplicate copies of all your diskettes. This will prevent the loss of your programs and data due to a power blackout, coffee spilled on a diskette, or other such accidents. A copy of a diskette, file, or program is called a **backup**.

You should also make a copy of the System Master Diskette the first time you use it, and then use only the copy to boot the disk. The original diskette should be stored in a safe place so that yet another copy can be made if the first copy is damaged. Here is the procedure for making a backup copy of a diskette. These instructions are for single disk drive systems. For systems with two drives see the comment below.

1. Load the System Master Diskette into drive 1 and boot the disk as outlined in Section 1.5.
2. Type

   ```
   RUN COPYA
   ```

3. The computer will display

   ```
   APPLE DISKETTE DUPLICATION PROGRAM
   ORIGINAL SLOT: DEFAULT=6
   ```

   Press

   ```
   RETURN
   ```

   The computer will then display

```
DRIVE: DEFAULT=1
```

Again press

```
RETURN
```

This tells the computer that the disk from which you wish to copy will be in slot 6, drive 1. The default value is the value which the computer will assume unless you type in something different.

4. The computer will now display

```
DUPLICATE SLOT: DEFAULT=6
```

Press

```
RETURN
```

The display will now be

```
DRIVE: DEFAULT=2
```

Type

```
1
```

The 1 indicates the drive number which will be receiving the copy. (If you have two drives and wish to place the copy in drive 2, type 2.)

5. The computer will display:

```
-PRESS 'RETURN' KEY TO BEGIN COPY
```

If you wish to copy the System Master diskette, just press the **RETURN** key. To copy some other diskette, place the diskette to be copied in drive 1. If you have a second drive, place the duplicate in drive 2, and then press **RETURN**.

6. The computer will now copy from the original disk into RAM. If you have only one diskette drive, the computer will prompt you to insert the duplicate diskette in drive 1. After you insert the duplicate diskette, press the **RETURN** key. The data in RAM is now copied onto the duplicate diskette. It will be necessary for you to switch the original and duplicate diskettes several times. Just follow the instructions provided on the screen.

TEST YOUR UNDERSTANDING 1

Make a backup of the System Master diskette supplied with your diskette operating system.

A Word to the Wise. The backup procedure just described may be used to copy the contents of one diskette onto another diskette. Because diskettes are fragile, it is strongly suggested that you maintain duplicate copies of all your diskettes. A good procedure is to update your copies at the end of each session with the computer. This may seem like a big bother, but it will prevent

untold grief if, by some mishap, a diskette with critical programs or data is erased or damaged.

You should also consider covering the write protect notch on the source disk before copying. This will prevent accidental destruction of files on this disk.

Note that you may not be able to make copies of diskettes which you purchased containing programs which someone else wrote. In fact, trying to do so may destroy the original contents of the diskette. These programs are specially recorded to protect the copyright. Always be careful to read the furnished documentation before attempting to make copies of such diskettes.

The Current Disk Drive

One of the diskette drives is designated the **current diskette drive**. If you give a DOS command without specifying which diskette drive, DOS will automatically assume that you are referring to the current diskette drive. The start-up sequence will designate drive 1 as the current drive. As soon as you give a DOS command specifying a drive other than the current drive, the drive you specify becomes the current drive.

Suppose, for example, that the current diskette drive is 1 and that we give the command

```
SAVE INVENTORY
```

The program which is currently in RAM will then be saved on the diskette in drive 1 under the name INVENTORY. If you then give a DOS command which involved drive 2, the current drive will change to drive 2.

Catalog

The DOS command **CATALOG** allows you to ask for the directory listing the names of all files on a given diskette. For example, to display the directory of the diskette in drive 1, just type

```
CATALOG,D1
```

followed by **RETURN**. (You may omit the D1 if drive 1 is the current drive.)

On the Apple IIe, you may obtain a catalog listing for any drive which is attached by hitting the ESC key and then the drive number. For example, hitting the ESC key and then 1 will give you a catalog of the files on the diskette which is in drive one.

Erasing Files

You may erase files from a diskette. To erase the file ROULETTE, type

```
DELETE ROULETTE
```

Be careful when you use the delete command. You will no longer be able to access your file once it is deleted. This is another good reason for keeping a backup disk.

Renaming a File

You may rename a file by using the **RENAME** command. To change the name of ROULETTE to GAME, we use the command

```
RENAME ROULETTE,GAME
```

Note that the old name always comes **first**, followed by the new name.

Locking and Unlocking Files

When you have written a program and it is running to your satisfaction, you want to be sure that you don't inadvertently overwrite it, replacing your good version with another. You can protect any individual file on your disk by LOCKing it. This allows you to read and run the program but will not allow you to write anything to a file of this name. To lock your program files simply type

```
LOCK name
```

where name is the name of the file.

You can unlock the file at any time by using the command

```
UNLOCK name
```

When you list the names of programs on a diskette using the CATALOG command, the names of the locked files will be preceded by an asterisk.

Reprise

In this chapter, we have covered the fundamental procedures you will use to write and run programs on the Apple II Plus and IIe computers. Before proceeding, check that you understand each of the following steps:

1. Starting your computer, including "booting" the disk if you have a disk drive.
2. Typing a program into RAM via the keyboard. This includes the use of the NEW command, correcting mistakes by retyping a line, and LISTing the program to see what is contained in RAM.
3. Executing a program by typing in the word RUN.
4. Saving the program on diskette.
5. Loading a program back into RAM from diskette.

If you have mastered these fundamental operations, you are ready to begin writing your own BASIC programs. The next couple of chapters will introduce you to the rules for Applesoft BASIC as well as some tips for developing good programs.

3
Beginning BASIC

3.1 Giving Names to Numbers and Words

We've already discussed the idea that the computer is not a "brain"; it cannot think. All the computer can do is follow the instructions we provide. Those instructions must be explicitly written so the computer can clearly interpret them. In this chapter we will begin to learn the rules for constructing programs, or sets of instructions, for the computer.

Before we proceed to the specific rules, let's discuss the general idea behind programs. If all the computer can do is follow the instructions we carefully devise, you may wonder what makes the computer such a powerful tool.

The primary advantage the computer has over the human mind is speed. If we can set up instructions for the computer to solve a problem, the computer can run through those instructions much faster than an average human being can. If we wanted to find what our average monthly utility bill was, for example, the computer could add the monthly totals and find the average more quickly than we could.

The difficulty is that in order to get the computer to perform the arithmetic for us, we would first have to write the instructions. That could be somewhat time consuming. However, if we write a general set of instructions for finding the average of 12 items, we could use the same program over and over again for different items, including the average monthly utility bill, the average monthly phone bill, the average number of sales per month, the average monthly revenue, and so on. We could make the instructions even more general so that we could change the number of items averaged. Then we could use the same set of instructions for finding the average of any data.

In general, the value of writing a computer program is to have a general set of instructions which can be used repeatedly. We gained some insight into how this might be done in the program to convert feet to meters shown in Figure 2-1. We used the symbols FT and MT to stand for feet and meters, respectively. Every time we ran the program we could assign different numbers to these symbols, thus making our set of instructions general enough to handle any conversion of feet to meters. In the language of computer programming, these symbols are called *variables.* We can write the set of instructions using these symbols as long as we tell the computer what values to assign to them before we try to use them in any computations. In the TEST

YOUR UNDERSTANDING exercise of Section 2.4 (page 32), N1 and N2 were used to stand for the two numbers we wished to add, and S was used to stand for the sum of those two numbers.

A numeric variable is a symbol used to represent a number.

In developing a program it is important to distinguish between variables and constants. A *numeric constant* is a number which appears in a program in its numeric form. Some numeric constants are .3408, −5.786, 2.5, and 1324. If a number remains the same each time the set of instructions is executed, then you include that number in the program in its numeric form. An example of a numeric constant is the .3048 factor used in the FEET-TO-METERS program.

In certain applications you may wish to specify your numeric constants in *exponential format.* This will be especially helpful in the case of very large and very small numbers. Consider the number 15,300,000,000. It is very inconvenient to type all the zeros. This large number can be written in handy shorthand as 1.53E+10. The 1.53 indicates the first three digits of the number. The E+10 means that you move the decimal point in the 1.53 to the right 10 places. Similarly, the number −237,000 may be written in the exponential format as −2.37E+05. The exponential format may also be used for very small numbers. For example, the number .00000000054 may be written in exponential format as 5.4E-10. The −10 indicates that the decimal in 5.4 is to be moved 10 places to the *left.*

TEST YOUR UNDERSTANDING 1 (answers on page 43)

a. Write these numbers in exponential format: .00048 , −1374.5
b. Write these numbers in decimal format: −9.7E+03, 9.7E−03, −9.7E−03

If a number is likely to change each time the program is run, or even within the same execution of the program, then it should be made a variable. In Applesoft BASIC *numeric variables,* or symbols used to represent numbers, must begin with an alphabetic character. Up to 237 alphabetic characters or numbers may follow that first character. You cannot use any special characters like hyphens, slashes, or periods in variable names. Thus Applesoft BASIC will recognize the following as valid names for numeric variables:

R2, D2, R2D2, GROSS, NET, FT, MT, A, B, S, BILL

The following, however, are not valid:

R-2, GROSS.PAY, PRICE/UNIT

Word of Warning. Even though Applesoft seems to accept 238 characters in names of numeric variables, it actually looks at only the first two characters.

Thus, Applesoft will not distinguish between the two variables R2 and R2D2 because they both begin with the same two characters. Similarly, a variable called PRICE will not be differentiated from a variable called PROFIT because they both begin with PR. The mnemonic value of using longer variable names in a program is somewhat offset by the additional care which must be taken to ensure that each variable is uniquely named. For this reason, the programs which we develop in this book will use variable names which are no more than two characters in length.

The numeric variables we have been discussing so far have been flexible enough to handle any number which we might want to use. These are frequently called *real variables*. Sometimes, however, we are interested in using numeric variables which never have any fractional parts; they have no decimal places. Such numeric variables are called *integer variables.*

In Applesoft BASIC we can denote an integer variable (a variable which can only take on whole-number values) by following the variable name with a percent sign. In this case, the special character has a specific meaning and can appear only at the end of the variable name.

For example, P1 is the name of a numeric variable which can take on fractional values, but P1% is the name of an integer variable and can only take on whole numbers between -32767 and +32767. Some valid integer variable names are:

R1%, APPLE%, N%

Some names which are not valid are:

5F%, APPLE-PIE%, E.T%

So far we have been discussing numbers, but there are many important situations where the things with which we are working are really strings of letters, special symbols, and numbers not to be used in calculations. Some examples are names, phone numbers, or social security numbers. These are called *strings*. String constants, which do not change from one execution of the program to the next, are placed in quotes. They are sometimes referred to as *literals*.

We had an example of a string constant in the FEET-TO-METERS program when we used the instruction

```
10 PRINT "PROGRAM TO CONVERT FEET TO METERS"
```

The words in quotes form the string constant or literal. Each time the program is executed, the same phrase will be printed out.

In some applications we wish to change the string each time. For example, if we want to process a payroll, the name of each employee will be different. In this case we need some symbol to represent the string when we write the general set of instructions. This symbol is called a *string variable*. String variables are distinguished from numeric variables by the fact that they end with a dollar sign. Some valid string variable names are:

N$, P1$, PHONE$

Again, because Applesoft only looks at the first two characters, we will name strings with no more than two characters followed by a dollar sign.

In summary, we have three kinds of variables in Applesoft BASIC: *real numeric variables, integer numeric variables,* and *string variables.* We also have *numeric constants* which appear in the program in numeric form, and *string constants* which appear in the program enclosed in quotes. We will see examples of all of these in the next section as we discuss how to get data into and out of the computer.

Assigning Values to Variables

Now that we have numeric and string variables, we need to know how to assign values to these variables. The easiest way to give a variable a value is to use a BASIC *assignment statement.* Some examples of these are

```
A=4
X$="HELLO THERE"
```

Such statements are called assignment statements because they take the value which is on the right of the equal sign and assign it to the variable on the left. Sometimes you will see assignment statements written with the word **LET** in front of the variable name. This is permissible in Applesoft BASIC, though it is not required. Because we can save a great deal of typing time by omitting this word, we will not include it. We will look more closely at assignment statements when we discuss how to do arithmetic in BASIC.

Exercises (answers on page 310)

1. Indicate whether each of the following is a valid variable name. If so, indicate the type of value it represents. If not, indicate why not.

   a. `TR$`
   b. `APPLE`
   c. `L5`
   d. `Y*Y`
   e. `5G`
   f. `Z%`
   g. `NAME$`
   h. `PR`

Convert the following numbers to exponential format.

2. 23,000,000
3. 175.25
4. −200,000,000
5. .00014
6. −.000000000275
7. 53,420,000,000,000,000

Convert the following numbers in exponential format to standard format.

8. 1.59E5
9. −20.3456E6

10. −7.456E−12
11. 2.39456E−18

What is wrong with the following Applesoft BASIC statements?

12. `10 LET A="YOUTH"`
13. `10 LET AA=-12`
14. `10 LET A$=57`
15. `LET ZZ$=Address`
16. `250 LET AAA=-9`
17. `10000 LET 1A=-2.34567`

ANSWERS TO TEST YOUR UNDERSTANDING 1

a. 4.8E−4, −1.3745E3
b. −9700, .0097, −.0097

3.2 Some Simple Printing Programs

At this point we know how to write simple assignment statements in BASIC by placing the variable name on the left of an equality sign and the value we wish to give that variable on the right. To be complete, the statement would need a line number.

Let us look again at the Applesoft BASIC program in Section 2.1.

```
10 PRINT "PROGRAM TO CONVERT FEET TO METERS"
20 INPUT "NUMBER OF FEET? ";FT
30 MT=.3048 * FT
40 PRINT "EQUIVALENT # OF METERS=";MT
50 END
```

This program illustrates two very important features common to all Applesoft BASIC programs:

1. The instructions of a program must be numbered. Each line must start with a line number. The computer executes instructions in order of increasing line number.
2. The **END** instruction identifies the end of the program. On encountering this instruction, the computer stops running the program and displays], the Applesoft BASIC prompt. Although in Applesoft BASIC the computer will recognize there are no more instructions and stop execution even without an **END** statement, there will be problems later in the book where we will want to stop the execution at different points in the program. In these cases the **END** statement will be very important.

Note that line numbers need not be consecutive. For example, it is perfectly acceptable to have a program whose line numbers are 10, 23, 47, 55, or 100. Also note that it is not necessary to type instructions in their numerical order. You could type line 20 and then go back and type line 10. The computer will sort out the lines and rearrange them according to increasing num-

bers. This feature is especially helpful in case you accidentally omit a line while typing your program.

Another important fact about line numbering is that if you type two lines having the same line number, the computer erases the first version and remembers the second version. This feature is very useful for correcting errors; if a line has an error, just retype it!

This means that if we wanted to use our simple assignment statements in a program, we would need to give each statement a line number. Even with this we would be very limited because we have not yet discussed any way for the computer to display data. We will discuss this in detail in Section 3.4, but in order to allow us to start writing programs, let's learn how to use the **PRINT** instruction to display numeric and string constants and variables.

For example, consider this instruction:

```
10 PRINT "PATIENT HISTORY"
```

During program execution, this statement will create the following display:

```
PATIENT HISTORY
```

Type **NEW** then type in this line and RUN. See that the computer does indeed produce this result. In this example we have used the **PRINT** statement to display a string constant on the screen. We could also write a similar instruction to display a numeric constant. For example:

```
20 PRINT 5.6
```

This program will display the following:

```
5.6
```

Type NEW again and then type in line 20 above and see that the computer displays the desired result.

In order to display several numeric and string constants on the same line, separate them by commas in a single **PRINT** statement. For example, the instruction

```
10 PRINT "AGE","BIRTHPLACE"
```

will cause two words to be printed:

```
AGE         BIRTHPLACE
```

An example combining both numeric constants and string constants in a single **PRINT** statement is:

```
100 PRINT "AGE", 65.43
```

How does the computer determine the spacing on a line? Each line is divided into **print zones**. Two print zones consist of 16 spaces and one has eight. By placing a comma in a PRINT statement, you are telling the computer to start the next string of text at the beginning of the next print zone. That means the two words above, AGE and BIRTHPLACE, begin in columns 1 and 16, respectively (see Figure 3-1). Note that the third print zone (positions 33-40) is available only if nothing is printed in positions 24-32. Thus, if

we wanted to print out AGE, BIRTHPLACE, and GENDER, we would not be able to fit it on one line by printing this way. We could, however, write it as BIRTHPLACE, AGE, and GENDER. Moreover, to use print zone 2 you must leave print position 32 blank.

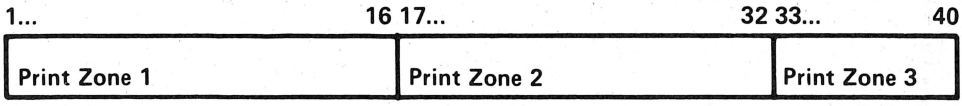


Figure 3-1. Print zones.

Example 1. Suppose that a retail store selling office furniture sells a chair and a desk. The chairs cost $75.00 and the desks cost $250.00, for a total sale of $325.00. There is a 5% tax on the sale, or a tax of $16.25. Thus, the customer's total bill is $341.25. Prepare a bill for the customer.

Solution. Let us insert two headings on our bill: Item and Cost. We then print two lines, corresponding to the two types of items purchased, and a line for the tax. Finally, we print the total due as shown here.

```
10 PRINT "ITEM","COST"
20 PRINT
30 PRINT "CHAIR",75.00
40 PRINT "DESK", 250.00
50 PRINT
60 PRINT "TAX ",16.25
70 PRINT
80 PRINT "TOTAL DUE",341.25
90 END
```

Note the **PRINT** statements in lines 20, 50, and 70. They specify that a blank line is to be printed. We could also specify a blank line by using

```
PRINT " "
```

If you are planning to use a printer with your system, you may wish to use this form. Many printers will not print a blank line by simply saying PRINT. If we now type **RUN** (followed by **RETURN**), the screen will look like this:

```
RUN

 ITEM            COST

 CHAIR           75.00
 DESK            250.00

 TAX             16.25

 TOTAL DUE       341.25
```

You may think that the above invoice is somewhat sloppy because the columns of figures are not properly aligned. Patience! We will learn to align the columns after we have learned a bit more programming.

TEST YOUR UNDERSTANDING 1 (answer on page 47)

Write a program to print the following display.

```
                NAME

LAST            FIRST

SMITH           JOHN
```

We can also use the **PRINT** statement to write out the values of numeric and string variables in much the same way. For example, consider the following program:

```
10 T$="TOTAL DUE"
20 T=341.25
30 PRINT T$,T
40 END
```

If we type this program in and then type **RUN** (followed by RETURN), the results will be

```
TOTAL DUE        341.25
```

This is the same result we obtained from line 80 in the previous program. In fact, we can obtain this same line in still another way:

```
10 T=341.25
20 PRINT "TOTAL DUE",T
30 END
```

Type this program in and **RUN** it to verify that you obtain the same output.

We can now write simple programs using the assignment statement and the **PRINT** statement. In the next section we will learn how to expand our capabilities by letting the computer do some arithmetic for us.

Deferred and Immediate Modes

When writing a BASIC program, we always precede each line with a line number. In this case the computer does not actually execute the instructions until we say **RUN**. This is called **deferred execution mode** because execution is deferred until we say RUN.

It is also possible to write BASIC instructions in immediate mode. In this case they are written without line numbers. As soon as the RETURN key is pressed, the computer executes the instruction. The instruction is not saved for re-execution later. The immediate mode can be helpful if you want to check out some BASIC statement without waiting until the entire program is written.

Exercises (answers on page 310)

Describe the output from each of these programs.

1.
```
10 PRINT "JONES","MARY ELIZABETH"
20 END
```
2.
```
10 PRINT "SILVER","GOLD","COPPER"
20 PRINT 327,448,1052
30 END
```
3.
```
10 PRINT , "FISH","MEAT"
20 PRINT "MON", "1,245","2,348"
30 PRINT "TUE", " 248","3,459"
40 END
```

ANSWER TO TEST YOUR UNDERSTANDING 1

```
10 PRINT ,"NAME"
20 PRINT
30 PRINT "LAST","FIRST"
40 PRINT
50 PRINT "SMITH","JOHN"
60 END
```

3.3 Doing Arithmetic in BASIC

In our initial discussion of assignment statements in Section 3.1, all we did was assign a constant to a variable. In the first example we took a numeric constant and assigned it to a numeric variable. Instead of letting the right-hand side be a simple constant, we can let it be an arithmetic expression combining the values of several arithmetic constants and variables. Your Apple II Computer will perform all the standard calculations that can be done with a calculator.

Most arithmetic operations are written in the standard fashion. For example, addition and subtraction are written for the computer in the usual way:

```
5 + 4, 9 - 8
```

Multiplication is typed using the asterisk symbol *, which shares the ":" key. For example, the product of 5 and 3 is typed as

```
5*3
```

Division is typed using a slash (/). For example, 8.2 divided by 15 is typed as

```
8.2/15
```

Example 1. Write an Applesoft BASIC program to calculate the sum of 54.75, 78.83, and 548, assign it to the variable S1, and print it out.

Solution. One possible BASIC program for this problem is

```
10 S1=54.75 + 78.83 + 548
20 PRINT S1
30 END
```

TEST YOUR UNDERSTANDING 1 (answer on page 51)

Write a computer program which creates the following display.

```
                    BUDGET-APRIL
FOOD                   387.50
RENT                   475.00
CAR                    123.71
UTILITIES              146.00
ENTERTAINMENT          100.00
TOTAL           (Calculate total)
```

Applesoft BASIC carries out arithmetic operations in a special order. It scans an expression and carries out all multiplication and division operations first, *proceeding in left-to-right order*. It then returns to the left side of the expression and performs addition and subtraction, also in a left-to-right order. If parentheses occur, these are evaluated first following the same rules stated above. If parentheses occur within parentheses, the innermost parentheses are evaluated first.

In summary, the arithmetic expression in the innermost parentheses is evaluated first. In evaluating an arithmetic expression, the computer performs all multiplication and division before it performs addition and subtraction. If an arithmetic expression contains more than one operation of the same priority, the operations are performed in order from the left to the right.

Example 2. What are the numerical values which Applesoft BASIC will calculate from these expressions?

a. (5 + 7)/2
b. 5 + 7/2
c. 5 + 7*3/2
d. (5 + 7*3)/2

Solution.

a. The computer first applies its rules for the order of calculation to determine the value in the parentheses, namely 12. It then divides 12 by 2 to obtain 6.
b. The computer scans the expression from left to right performing all multiplication and division in the order enccuntered. First it divides 7 by 2 to obtain 3.5. It then rescans the line and performs all additions and subtractions in order. This gives

   5 + 3.5=8.5

c. The computer first performs all multiplication and division in order:

5 + 10.5

Now it performs addition and subtraction to obtain 15.5.

d. The computer calculates the value of all parentheses first. In this case it computes 5 + 7*3=26. (Note that it does the multiplication first!) Next it rescans the line which now looks like this:

26/2

It performs the division to obtain 13.

TEST YOUR UNDERSTANDING 2 (answer on page 51)

Calculate 5+3/2+2 and (5+3)/(2+2) as the computer would.

Example 3. Write an Applesoft BASIC program to calculate and print the quantity

$$\frac{22\times 18 + 34\times 11 - 12.5\times 8}{27.8}$$

Solution. The BASIC program with the answer assigned to the variable EX is:

```
10 EX=(22*18 + 34*11 - 12.5*8)/27.8
20 PRINT EX
30 END
```

Note that we used parentheses. They tell the computer that the entire quantity in parentheses is to be divided by 27.8. If we had omitted the parentheses, the computer would divide −12.5*8 by 27.8 and add 22*18 and 34*11 to the result.

TEST YOUR UNDERSTANDING 3 (answers on page 51)

Write Applesoft BASIC statements to calculate:
a. ((4x3 + 5x8 + 7x9)/(7x9 + 4x3 + 8x7)) x 48.7
b. 27.8 % of (112 + 38 + 42)
c. The average of the numbers 88, 78, 84, 49, 63

Exponentiation

There is another common arithmetic operation which we have not yet mentioned: *exponentiation*. Suppose that A is a number and n is a positive whole number (this means that n is one of the numbers 1,2,3,4, . . .). Then *A raised to the nth power* is the product of A times itself n times. This quantity is usually denoted in mathematics texts as A^n, and the process of calculating it is called *exponentiation*. For example

$2^3 = 2*2*2 = 8$, $5^7 = 5*5*5*5*5*5*5 = 78125$

It is possible to calculate A^n by repeated multiplication, but if n is large, this can be tiresome to type. Applesoft BASIC provides a shortcut for typing this function. Exponentiation is denoted by the symbol ^, which is produced by hitting the key with the upward-pointing arrow (this symbol shares the "N" key on the bottom of the keyboard). For example, 2^3 is denoted 2^3.

The operation of exponentiation takes precedence over multiplication and division. This is illustrated in the following example.

Example 4. Determine the value which Applesoft BASIC assigns to this expression

```
20*3 - 5*2^3
```

Solution. The exponentiation is performed first to yield

```
20*3 - 5*8=60 - 40
=20
```

In using exponentiation, it is also possible to have fractional powers. For example, the square root of X is $X^{1/2}$. In BASIC you would write this as:

```
X^(1/2)
```

In summary, the rules for evaluation of BASIC expressions are:

- **Rule 1:** The arithmetic expression in the innermost parentheses is evaluated first.
- **Rule 2:** In evaluating an arithmetic expression, the computer performs operations according to the following priorities:

  1. all exponentiation is done first
  2. all multiplication and division is performed next
  3. addition and subtraction operations are last.

- **Rule 3:** If an arithmetic expression contains more than one operation of the same priority, the operations are performed in order from the left to the right.

TEST YOUR UNDERSTANDING 4 (answers on page 51)

Evaluate the following first manually and then by an Apple II Computer program.

a. $2^4 \times 3^3$
b. $2^2 \times 3^3 - 12^2/3^2 \times 2$
c. (2 squared times 3 cubed - 12 squared)/(3 squared x 2)

Notice that c has parentheses where b does not. This will change the way the computer evaluates the expressions.

Exercises (answers on page 310)

Write Applesoft BASIC programs to calculate the following quantities.

1. $57 + 23 + 48$
2. $57.83 \times (48.27 - 12.54)$
3. $127.86/38$
4. $365/.005 + 1.02^5$

In Exercises 5-10, determine the output of the given program.

5.
```
10 LET A=5:B=5
20 PRINT A + B
30 END
```

6.
```
10 LET AA=5
20 PRINT AA*B
30 END
```

7.
```
10 LET A1=5
20 PRINT A1^2 +5*A1
30 END
```

8.
```
10 LET A=2: B=7: C=9
20 PRINT A+B, A-C, A*C
30 END
```

9.
```
10 LET A$="JOHN JONES"
20 LET B$="AGE": C=38
30 PRINT A$, B$, C
40 END
```

10.
```
10 LET X=11, Y=19
20 PRINT 2*X
30 PRINT 3*Y
40 END
```

11. Mrs. Anita Smith went to her doctor with a broken leg. Her bill consists of $45 for removal of the cast, $35 for therapy, and $5 for drugs. Her major medical policy will pay 80 percent directly to the doctor. Use the computer to prepare an invoice for Mrs. Smith.
12. Consider the numbers 2.3758, 4.58321, 58.11. Write a program which computes their sum, product, and the sum of their squares.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1,2,3, and 4

1:
```
10 PRINT ,"   BUDGET-APRIL"
20 PRINT "FOOD", 387.50
30 PRINT "RENT",  475.00
40 PRINT "CAR",  123.71
50 PRINT "UTILITIES", 146.00
60 PRINT "ENTERTAINMENT", 100.00
70 PRINT , "------"
80 T=387.50+475.00+123.71+146.00+100.00
85 PRINT "TOTAL",T
90 END
```

2: `5+3/2+2=5 +1.5 +2=8.5, (5+3)/(2+2)=8/4=2`

3: a.
```
10 PRINT ((4*3+5*8+7*9)/(7*9+4*3+8*7))*48.7
20 END
```

b.
```
10 PRINT .278*(112+38+42)
20 END
```

c.
```
10 PRINT (88+79+84+49+63)/5
20 END
```

4: a. 432
b. 76
c. -2

3.4 Displaying and Inputting Data

In this section we will discuss a few variations on the **PRINT** statement which will give you a little more flexibility in deciding how to display information on the screen. Let us begin by reviewing what we have already learned about printing.

As we discussed in Section 3.2, there are 40 print positions in each line. These are divided into print zones of 16 characters each with only eight characters in the third print zone. To start printing at the beginning of the next print zone, we insert a comma between the items to be printed. Because of the way that Applesoft works, the only way that you can print starting in the third print zone is by leaving most of the second zone blank. This requires us to be very careful about the data we print out. We can expand our capability by using the semicolon to separate printing fields. If we separate items to be printed by semicolons, we eliminate all blank spaces between them. For example:

```
10 A=5
20 PRINT "THE VALUE OF A IS EQUAL TO ";A
30 END
```

Running this will result in the following screen display:

```
THE VALUE OF A IS EQUAL TO 5
```

Note that the first print item ends with a space. This is to guarantee a space between the final O in TO and the 5.

If we use the semicolon, we must write the programs so that we deliberately leave spaces where we want them. For example, consider the following program:

```
10 A=10
20 B=20
30 C=40
40 PRINT A;B;C
```

Running this program will result in the following output:

```
102040
```

All of the values are run together, making them difficult to read. To separate the numbers, we can place blank spaces between the values by instructing the computer to print out string constants which are blanks like this:

```
10 A=10
20 B=20
30 C=40
40 PRINT A;" ";B;" ";C
```

Running this program will result in the following output:

```
10 20 40
```

By making the string constant with more than one blank we can leave more spaces between the values.

There is another way to leave spaces between values. This is by using the **SPC** phrase in the print statement. For example, by saying **SPC(3)** we tell the computer to leave three spaces. The program

```
10 A=10
20 B=20
30 C=40
40 PRINT A;SPC(3);B;SPC(3);C
```

produces this output

```
10   20   40
```

Note that we can achieve the same effect by using a string constant with three blanks or by using **SPC(3)**. We can only use the **SPC** phrase in the **PRINT** statement.

It is also possible to do arithmetic within the **PRINT** statement. For example,

```
10 PRINT A/B
```

would cause the value of A to be divided by the value of B and then the answer would be printed out.

Applesoft BASIC also allows a short cut in entering **PRINT** statements. Instead of typing in the word **PRINT**, you can just type in a question mark. The computer will interpret this as a **PRINT** command. Let's try this. Type in the following:

```
20 ? "HELLO"
```

When you list this line (**LIST**) you will see

```
20 PRINT "HELLO"
```

TEST YOUR UNDERSTANDING 1 (answer on page 59)

Write a program which allows you to set the variable A equal to 5 and the variable B equal to 7. The program should then display them as an addition problem in the form 5 + 7 = 12.

Horizontal Tabbing

You may begin a print item in any position on a line. To do this use the **TAB** command. A logical line may be up to 255 characters long. The positions of characters on logical lines are numbered from 1 to 255. An oversized line will wrap around to the next line.

The statement TAB(7) means to move to position 7. **TAB** is always used in conjunction with a **PRINT** statement. For example, the print statement

```
50 PRINT TAB(7);A
```

will print the value of the variable A beginning in print position 7.

It is possible to use more than one **TAB** per **PRINT** statement. For example, the statement

```
100 PRINT TAB(5);A;TAB(15);B
```

will print the value of A beginning in print position 5, and the value of B beginning in print position 15.

Note that we have used semicolons to separate each variable name, string constant, **SPC**, and **TAB**. It is not necessary to follow string constants, **SPC**, or **TAB** with semicolons. The statement will behave exactly as if the semicolon were present.

In typing a **PRINT** statement, you may run out of room on the line. To get around this problem, end the **PRINT** statement with a semicolon and continue the list of print items in another **PRINT** statement on the next line. For example, consider the pair of statements

```
100 PRINT "INVENTORY";
110 PRINT , "OF LADIES SHOES"
```

The first line has a single print item. The semicolon indicates continued printing on the same line. The comma which begins the second **PRINT** statement moves printing to the beginning of the next print zone, where the string in line 110 is printed. Here is what the output looks like:

```
INVENTORY                    OF LADIES SHOES
```

TEST YOUR UNDERSTANDING 2 (answer on page 59)

Write an instruction printing the value of A in column 25 and the value of B seven columns further to the right.

Inputting Data

So far we have been assigning values to variables by using assignment statements. Often, it is convenient to have the computer request data from you while the program is actually running. This can be accomplished via the **INPUT** statement. To see how this is done consider the statement

```
570 INPUT A
```

When the computer encounters this statement in the course of executing the program, it types out a ? and waits for you to respond by typing the desired value of A (and then hitting the **RETURN** key). The computer sets A equal to the numeric value you specified and continues running the program.

You may use an **INPUT** statement to specify the values of several different variables at one time. These variables may be numeric or string variables. For example, suppose that the computer encounters the statement

```
50 INPUT A,B,C$
```

It will type

```
?
```

You then type in the desired values for A,B, and C$ in the same order as they appear in the program, separated by commas. Suppose that you type

```
10.5, 11.42, BEARINGS
```

followed by a **RETURN**. The computer will then set

```
A=10.5, B=11.42, C$="BEARINGS"
```

If you respond to the above question mark by typing only a single number, 10.5 for example, the computer will respond with

```
??
```

to indicate that it expects more data. If you attempt to specify a string constant where you should have a numeric constant, the computer will respond with the message

```
? REENTER
```

and wait for you to repeat the **INPUT** operation.

It is helpful to include a prompting message which describes the input the computer is expecting. To do so put the message in quotation marks after the word **INPUT** and place a semicolon after the message (before the list of variables to be input). For example, when the computer encounters the program line

```
175 INPUT "ENTER COMPANY, AMOUNT "; A$, B
```

the dialog will be

```
ENTER COMPANY, AMOUNT AJAX OFFICE SUPPLIES, 2579.48
```

The underlined portion indicates your response to the prompt. The computer will now assign the values

```
A$="AJAX OFFICE SUPPLIES", B=2579.48
```

Note that if the string variable to be input contains commas, that string must be included in quotes. Otherwise the computer will interpret the comma as the end of the string and go on to the next variable. When writing a program which requires input of string values, be sure that the user understands this requirement. You may wish to include it in your prompt message or in a separate print statement before the input.

TEST YOUR UNDERSTANDING 3 (answer on page 59)

Write a program which allows you to set variables A and B to any desired values via an **INPUT** statement. Use the program to set A equal to 12 and B equal to 17. Then sum the two numbers and display the answer on the screen.

The **INPUT** statement instructs the computer to wait for the person sitting at the keyboard to type in the data values for the program. Still there are times when we might prefer to include the data within the program. We can do this with assignment statements but, as we shall soon see, this can limit our flexibility. Fortunately, Applesoft BASIC provides an alternate method for inputting data.

A given program may need many different numbers and strings. You may store this data in one or more **DATA** statements. A typical DATA statement has the form

```
10 DATA 3.457, 2.588, 11234, "WINGSPAN"
```

This data statement consists of four data items, three numeric and one string. The data items are separated by commas. You may include as many data items in a single **DATA** statement as the line allows. You may also include any number of **DATA** statements placed anywhere in the program although a common placement is at the end of the program.

Note we enclosed the string constant "WINGSPAN" in quotation marks although this is not necessary. A string constant in a **DATA** statement does not need quotes as long as the string does not include a comma. (A comma would signal the computer to end the string at that point.)

The **DATA** statements may be used to assign values to variables. You use one or more **READ** statements (in conjunction with the **DATA** statements). For example, suppose that the above **DATA** statement appeared in a program. Further, suppose that you wish to assign the values

```
A=3.457, B=2.588, C=11234, Z$="WINGSPAN"
```

This can be accomplished via the **READ** and **DATA** statements:

```
10 DATA 3.457, 2.588, 11234, "WINGSPAN"
100 READ A,B,C,Z$
```

On encountering a **READ** statement, the computer will look for a **DATA** statement. It will then assign values to the variables in the **READ** statement by taking the values from the **DATA** statement in the order in which they appear. If there is insufficient data in the first **DATA** statement, the computer will continue to assign values using the data in the next **DATA** statement. If necessary, the computer will proceed to the third **DATA** statement and so on until all the variables in the READ statement have been assigned values.

TEST YOUR UNDERSTANDING 4 (answer on page 59)

Assign the following values using a **READ** statement:

A = 5.1, B = 4.7, C = 5.8, D = 3.2, E = 7.9, F = 6.9

The computer maintains an internal pointer which points to the next **DATA** item to be used. If the computer encounters a second **READ** statement, it will start reading where it left off. For example, suppose that instead of the above **READ** statement, we use the two read statements

```
10 DATA 3.457, 2.588, 11234, "WINGSPAN"
100 READ A,B
200 READ C,Z$
```

Upon encountering the first statement, the computer will look for the location of the pointer. Initially, it will point to the first item in the first **DATA** statement. The computer will assign the values A = 3.457 and B = 2.588. The position of the pointer will be advanced to the third item in the **DATA** statement. Upon encountering the next **READ** statement, the computer will assign values beginning with the one designated by the pointer, namely C = 11234 and Z$ = "WINGSPAN".

TEST YOUR UNDERSTANDING 5 (answer on page 59)

What values are assigned to A and B$ by the following program?

```
10 DATA 10,30,"ENGINE", "TACH"
20 READ A,B
30 READ C$,B$
40 END
```

In certain applications you may wish to read the same **DATA** statements more than once. To do this you must reset the pointer via the **RESTORE** statement. For example, consider the following program:

```
10 DATA 2.3, 5.7, 4.5, 7.3
20 READ A,B
30 RESTORE
40 READ C,D
50 END
```

Line 20 sets A equal to 2.3 and B equal to 5.7. The **RESTORE** statement of line 30 moves the pointer back to the first item of data, 2.3. The **READ** statement of line 40 then sets C equal to 2.3 and D equal to 5.7. Without the **RESTORE** in line 30, the **READ** statement in line 40 would set C equal to 4.5 and D equal to 7.3.

There are two common errors in using **READ** and **DATA** statements. First, you may instruct the program to **READ** more data than is present in the **DATA** statements. Consider the following program:

```
10 DATA 1,2,3,4
20 READ A,B,C,D,E
30 END
```

This program attempts to read five pieces of data, but the **DATA** statement only has four. In this case, you will receive an error message

```
? OUT OF DATA ERROR IN 20
```

A second common error is attempting to assign a string value to a numeric variable. Such an attempt will lead to

```
? TYPE MISMATCH ERROR
```

The Remark Statement

It is very convenient to explain programs using remarks. For one thing, remarks make programs easier to read. Remarks also assist in finding errors and making modifications in a program.

To insert a remark into a program, we may use the REM statement. For example, consider the line

```
520 REM X DENOTES THE COST BASIS
```

Since the line starts with REM, it will be ignored during program execution. It is a non-executable statement.

Multiple Statements On A Single Line

It is possible to put several Applesoft BASIC statements on a single line. Just separate them by a colon. For example, instead of the two statements

```
10 A=5.784
20 B=3.571
```

we may use the single statement

```
10 A=5.784: B=3.571
```

To insert a remark on the same line as a program statement, use a colon followed by a **REM.**

```
10 A=PI*R^2 :REM A IS THE AREA,R IS THE RADIUS
```

The REM statement causes the remainder of the line to be ignored by the computer; therefore, if it is used on the same line with executable statements, it should always be placed at the end of the line. We will often place comments in the programs which follow so that they will be easier to understand.

TEST YOUR UNDERSTANDING 6 (answer on page 59)

What is the result of the following program?

```
10 A=7:B$="COST":C$="TOTAL"
20 PRINT C$,B$
30 PRINT "=",A
```

Exercises (answers on page 311)

1. Write a program similar to the FEET-TO-METERS program which we developed in Chapter Two to convert meters to feet (the conversion factor is 3.2808).
2. Write a program similar to the FEET-TO-METERS program which we developed in Chapter Two to convert liquid quarts to liters (the conversion factor is 0.9463).
3. Write a program similar to the FEET-TO-METERS program which we developed in Chapter Two to convert liquid liters to quarts (the conversion factor is 1.0567).

Write programs which generate the following displays. The lines of dots are not meant to be displayed. They are furnished for you to judge spacing.

4. THE VALUE OF X IS 5.378
. . . . . .

5. THE VALUE OF X IS 5.378
. . . . . .

6. Date 3/18/81

Pay to the Order of Wildcatters, Inc.
The sum of *********$89,385.00

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, 3, 4, 5, and 6

1:
```
10 A=5
20 B=7
30 S=A + B
40 PRINT A;" + ";B";"=";S
50 END
```

2:
```
10 PRINT TAB(25);A;TAB(32);B
```

3:
```
10 INPUT "TYPE IN TWO NUMBERS ";A,B
20 S=A + B
30 PRINT A;" + ";B";"=";S
40 END
```

4:
```
10 READ A,B,C,D,E,F
20 DATA 5.1,4.7,5.8,3.2,7.9,6.9
```

5:
```
A=10, B=30, C$="ENGINE", B$= "TACH"
```

```
 6:  TOTAL        COST
     =              7
```

3.5 Letting Your Computer Make Decisions

Up to this point the computer has always executed our statements in sequence, starting from the top of the program and proceeding, one after the other, to the end of the program. There are many occasions, however, when it is desirable to alter the normal sequence of program line execution.

Just such an occasion might be when we wish to instruct the computer to return to a previous statement and repeat the sequence of steps. For these applications we may use the **GOTO** instruction. This instruction has the form

```
GOTO < line number >
```

For example, the instruction

```
1000 GOTO 300
```

will send the computer to line 300 for its next instruction.

Example 1. A firm keeps records on its employees. The data kept includes the employee's name, gross pay for each pay period, and the tax rate for that individual (obviously this is a highly simplified example). Write a program which will read in the data for each employee one after the other, calculate the net pay and write out the employee's name and net pay.

Solution. Let us denote the employee's name as N$, the gross pay as GP, the tax rate as TX, and the net pay as NT. Employee data will be kept in DATA statements.

```
5 PRINT " NAME";TAB(25);"NET PAY"
10 READ N$,GP,TX
20 NT=GP * (1 - TX)
30 PRINT N$;TAB(25);NT
40 GOTO 10
50 END
100 DATA MARY JONES,345.00,.20
110 DATA PETER PAUL,310.00,.15
```

Note we have included only two data lines, but in practice you would have one line for each employee. If you type this program in and run it, you will obtain the following output:

```
   NAME                    NET PAY
MARY JONES                   276
PETER PAUL                   263.5
?OUT OF DATA ERROR IN 10
```

The error message occurs because when we send the computer back to statement 10 for the third time, we will not have any more data left.

We can correct this problem by using a conditional branch statement. This statement allows you to branch to another instruction depending upon the truth or falsity of some condition.

The vehicle which Applesoft BASIC uses to do this is the **IF...THEN** statement. The **IF** part of such a statement allows us to ask a question. If the answer is YES, then the computer carries out the **THEN** part of the statement. If the answer is NO, then the computer goes on to the next statement in numerical order.

For example, consider the statement

```
500 IF N=0 THEN GOTO 300
```

First, the computer determines if N is equal to zero. If so, it proceeds to statement number 300 and resumes execution from that point. If N is not zero, then the computer goes directly to the next instruction after line 500 and continues program execution from that instruction. Notice the **GOTO** after **THEN**. This may be omitted. That is, the statement may be written as

```
500 IF N=0 THEN 300
```

In any case, putting the **GOTO** in will never hurt anything.

After **IF** you may insert any expression which the computer may test for truth or falsity. Some examples are:

N=0
N > 5 (N is greater than 5)
N < 12.9 (N is less than 12.9)
N >=0 (N is greater than or equal to 0)
N <=-1 (N is less than or equal to -1)
N >< 0 (N is not equal to 0)
A + B <> C (A + B is not equal to C)
A^2 + B^2 <=C^2 ($A^2 + B^2$ is less than or equal to C^2)

Example 2. Modify the program in Example 1 so that the **OUT OF DATA** error message is eliminated.

Solution. We will accomplish this by putting in a dummy set of data and then testing to see whether the name we have just read in corresponds to the dummy value. In our case we will let this dummy name (sometimes also called *flag* or *sentinel data* because it flags the end of the data) be "NOMORE". When choosing sentinel values always be sure that they do not correspond to any valid data values. In this case we are assuming that no employee has a name like NOMORE. Because we are asking the computer to READ three data items, the flag data must also include three dummy values.

```
5 PRINT " NAME";TAB(25);"NET PAY"
10 READ N$,GP,TX
15 IF N$="NOMORE" THEN 50
20 NT=GP * (1 - TX)
30 PRINT N$;TAB(25);NT
40 GOTO 10
```

```
50 END
100 DATA MARY JONES,345.00,.20
110 DATA PETER PAUL,310.00,.15
900 DATA NOMORE,0,0
```

We added a final data statement containing the dummy data at line 900. Then at line 15 we tested to see whether the name which we had read in was our flag value "NOMORE". If it was we jumped to statement 50 which ended the execution.

TEST YOUR UNDERSTANDING 1 (answer on page 70)

Consider the following sequence of instructions.

```
100 IF A>=5 THEN GOTO 200
110 IF A>=4 THEN GOTO 300
120 IF A>=3 THEN GOTO 400
130 IF A>=2 THEN GOTO 500
```

Suppose that the current value of A is 3. List the sequence of line numbers which will be executed.

The form of the **IF...THEN** statement we have been using is a special case. Any executable statement in BASIC can follow the **THEN**. Consider

```
500 IF N=0 THEN PRINT "CALCULATION DONE"
```

In this case, if the value of N is 0, then the computer will print out "CALCULATION DONE" and proceed to the next statement after line 500. If the statement is not true, then it will merely go immediately to the line following 500 without printing out anything.

TEST YOUR UNDERSTANDING 2 (answers on page 70)

Write instructions which do the following:

a. If A is less than B, then print the value of A plus B; if not, then go to the end.
b. If $A^2 + D$ is at least 5000 then go to line 300; if not, go to line 500.
c. If N is larger than or equal to the sum of I and K, then set N equal to the sum of I and K; otherwise, let N equal K.

NOTE: Any statement which follows on the same line as an **IF...THEN** statement will be executed only if the **THEN** part is executed. For example, in the statement

```
10 IF A < B THEN C=D: GOTO 300
20 C=E
```

the computer will do the following: if A is less than B, the computer will set C equal to D. It will then go to 300. On the other hand, if A is not less than B, the computer will go to the statement in line 20.

The following examples illustrate some of the uses of the **IF. . .THEN** and **GOTO** statements.

Example 3. A lumber supply house has a policy that a credit invoice may not exceed $1,000, including a 10 percent processing fee and 5 percent sales tax. A customer orders 150 2x4 studs at $1.99 each, 30 sheets of plywood at $14.00 each, 300 pounds of nails at $1.14 per pound, and two double hung insulated windows at $187.95 each. Write a program which prepares an invoice and decides whether the order is over the credit limit.

Solution. Let's use the variables A1, A2, A3, and A4 to denote, respectively, the numbers of studs, sheets of plywood, pounds of nails, and windows. Let's use the variables B1, B2, B3, and B4 to denote the unit costs of these four items. The cost of the order is then computed as

```
A1*B1+A2*B2+A3*B3+A4*B4
```

We add 10 percent of this amount to cover processing and form the sum to obtain the total order. Next we compute 5 percent of the last amount as tax and add it to the total to obtain the total amount due. Finally, we determine if the total amount due is more than $1,000. If it is, we print out the message: ORDER EXCEEDS $1,000. CREDIT SALE NOT PERMITTED. Here is our program.

```
10  A1=150:A2=30:A3=300:A4=2
20  B1=1.99:B2=14:B3=1.14:B4=187.95
30  T=A1*B1+A2*B2+A3*B3+A4*B4
40 PRINT "TOTAL ORDER",T
50  P=.1*T
60 PRINT "PROCESSING FEE",P
70  TX=.05*(P+T)
80 PRINT "SALES TAX",TX
90 DU=T + P + TX
100 PRINT "AMOUNT DUE", DU
110 IF DU < 1000 THEN PRINT "CREDIT SALE OK":END
200 PRINT "ORDER EXCEEDS $1,000"
210 PRINT "CREDIT SALE NOT PERMITTED"
400 END
```

In this program we have made use of the fact that we can place several statements on one line if we separate them with colons. We use this in lines 10 and 20 to put the numerous assignment statements on one line.

Note the decision in line 110. If the amount due is less than $1,000, then the computer prints out a message okaying credit and then ends. On the other hand, if the amount due exceeds $1,000, the computer drops through to line 200 where credit is denied and the program ends.

Example 4. At $20 per square yard, a family can afford up to 500 square feet of carpet for their dining room. They wish to install the carpet in a circular

shape. It has been decided that the radius of the carpet is to be a whole number of feet. What is the radius of the largest carpet they can afford? (The area of a circle of radius "R" is PI times R^2, where PI equals approximately 3.14159 .)

Solution. Let us compute the area of the circle of radius 1,2,3,4,. . . and determine which of the areas are less than 500.

```
5 PRINT "THESE RADIUS VALUES YIELD AN AREA < 500"
10 PI=3.14159
20 R=1 : REM R IS THE RADIUS OF THE CIRCLE
30 A=PI*R^2 : REM A IS THE AREA OF THE CIRCLE
40 IF A>=500 THEN GOTO 100 : REM IF AREA IS AT LEAST 500,
    END
50 PRINT R : REM IF AREA IS LESS THAN 500, PRINT R
60 R=R + 1 : REM GO TO NEXT RADIUS
70 GOTO 30
100 END
```

Note line 40 contains an **IF. . .THEN** statement. If A, as computed in line 30, is 500 or more, then the computer goes to line 100, the **END**. If A is less than 500, the computer proceeds to the next line, namely 50. It then prints out the current radius, increases the radius by 1, and goes back to line 30 to repeat the entire procedure. Lines 30-40-50-60-70 are repeated until the area becomes at least 500. This set of instructions forms a loop. The computer decides the stopping point via the **IF. . .THEN** instruction.

Note also line 60. The arithmetic statement in that line would not be valid in mathematics because the left and right sides cannot be mathematically equal. This is a good example of the meaning of the equal sign in BASIC. Here the sign means to take the current value of the variable R, add one to it, and *assign* the new value to the variable R, thus incrementing R by one each time.

Example 5. A school board election has two candidates. The returns from the four wards of the town are as follows:

	Ward 1	Ward 2	Ward 3	Ward 4
Mr. Thompson	487	229	1540	1211
Ms. Wilson	1870	438	110	597

Calculate the total number of votes achieved by each candidate, the percentage achieved by each candidate, and decide who won the election.

Solution. Let A1, A2, A3, and A4 be the totals for Mr. Thompson in the four wards; let B1 through B4 be the corresponding numbers for Ms. Wilson. Let TA and TB denote the total votes for Mr. Thompson and Ms. Wilson respectively. Here is our program.

```
10  A1=487: A2=229: A3=1540: A4=1211
20  B1=1870: B2=438: B3=110: B4=597
30  TA=A1+A2+A3+A4 : REM TOTAL FOR THOMPSON
40  TB=B1+B2+B3+B4 : REM TOTAL FOR WILSON
50  T=TA + TB : REM TOTAL VOTES CAST
```

```
60  PA=100*TA/T : REM PERCENTAGE FOR THOMPSON
65 REM TA/T IS THE FRACTION OF VOTES FOR THOMPSON- MULTIPLY
66 REM BY 100 TO CONVERT TO A PERCENTAGE
70  PB=100*TB/T : REM PERCENTAGE FOR WILSON
110  A$="THOMPSON"
120  B$="WILSON"
130 REM 140-170 PRINT THE PERCENTAGES OF THE CANDIDATES
140 PRINT "CANDIDATE","VOTES","PCT"
150 PRINT A$,TA, PA
160 PRINT B$,TB, PB
170 REM 180-400 DECIDE THE WINNER
180 IF TA > TB THEN GOTO 300
190 IF TA < TB THEN GOTO 400
200 PRINT "TIE VOTE!"
210 GOTO 1000
300 PRINT A$, "WINS"
305 PRINT
310 GOTO 1000
400 PRINT B$, "WINS"
410 PRINT
1000 END
```

Note the logic used for deciding who won. In line 180 we compare the votes TA and TB. If TA is the larger, then A (Thompson) is the winner. We then go to 300, print the result, and END.

On the other hand, if TA > TB is *false*, then either B wins or the two are tied. According to the program, if TA > TB is false, we go to line 190, where we determine if TA < TB. If this is true, then B is the winner, we go to 400, print the result, and END.

If TA < TB is false, then the only possibility left is that TA = TB. According to the program, if TA = TB we go to 200, where we print the proper result, and then END.

TEST YOUR UNDERSTANDING 3 (answers on page 70)

Suppose that a credit card company charges 1.5 percent per month on any unpaid balance up to $500 and 1 percent per month on any excess over $500.

a. Write a program which computes the service charge and the new balance.
b. Test your program on the unpaid balances of $1,300 and $275.

The next three examples illustrate the use of the **IF...THEN** statement, and provide further practice in using the **INPUT** statement.

Example 6. You are a teacher compiling semester grades. Suppose there are four grades for each student and that each grade is on the traditional 0 to 100 scale. Write a program which accepts the grades as input, computes the semester average, and assigns grades according to the following scale:

90-100	A
80-89.9	B
70-79.9	C
60-69.9	D
< 60	F

Solution. We will use an **INPUT** statement to enter the grades into the computer. Our program will allow you to compute the grades of students, one after the other, via what is called a loop, i.e., the computer loops through the same set of instructions for each student. You may terminate the loop by entering a negative grade. Here is our program.

```
5 HOME
10 PRINT "ENTER STUDENT'S 4 GRADES."
20 PRINT  "SEPARATE GRADES BY COMMAS."
30 PRINT "FOLLOW LAST GRADE WITH RETURN."
40 PRINT "TO END PROGRAM, ENTER NEGATIVE GRADE."
50 INPUT A1,A2,A3,A4
60 IF A1 < 0 THEN END
70 IF A2 < 0 THEN END
80 IF A3 < 0 THEN END
90 IF A4 < 0 THEN END
100 A=(A1+A2+A3+A4)/4
110 PRINT "SEMESTER AVERAGE", A
120 IF A>=90 THEN PRINT "SEMESTER GRADE=A" : GOTO 10
130 IF A>=80 THEN PRINT "SEMESTER GRADE=B" : GOTO 10
140 IF A>=70 THEN PRINT "SEMESTER GRADE=C" : GOTO 10
150 IF A>=60 THEN PRINT "SEMESTER GRADE=D" : GOTO 10
160 PRINT "SEMESTER GRADE=F":GOTO 10
200 END
```

Note the logic for printing out the semester grades. First compute the semester average A. In line 120 we ask if A is greater than or equal to 90. If so, we assign the grade A, and go to the **END**. In case A is less than 90, line 120 sends us to line 130. In line 130, we ask if A is greater than or equal to 80. If so, then we assign the grade B. (The point is that the only way we can get to line 130 is for A to be less than 90. So if A is greater than or equal to 80, we know that A lies in the B range.) If not, we go to line 140 and so forth. This logic may seem a trifle confusing at first, but after repeated use it will seem quite natural.

Example 7. Write a program to maintain your checkbook. The program should allow you to record an initial balance, enter deposits, and enter checks. It should also warn you of overdrafts.

Solution. Let the variable B always contain the current balance in the checkbook. The program will ask for the type of transaction you wish to record. A "D" will mean that you wish to record a deposit; a "C" will mean that you wish to record a check; a "Q" will mean that you are done entering transactions and wish to terminate the program. After entering each transaction, the computer will figure your new balance, report it to you, will check for an

overdraft, and report any overdraft to you. In case of an overdraft, the program will allow you to cancel the preceding check!

```
10 INPUT "WHAT IS YOUR STARTING BALANCE"; B
20 INPUT "WHAT TRANSACTION TYPE (D,C,or Q)"; A$
30 IF A$="Q" THEN END
40 IF A$="D" THEN INPUT "DEPOSIT AMOUNT"; D:GOTO 110
100 IF A$="C" THEN 200
110 B=B + D : REM ADD DEPOSIT TO BALANCE
120 PRINT "YOUR NEW BALANCE IS", B
130 GOTO 20
200 INPUT "CHECK AMOUNT"; C
210 B=B - C : REM DEDUCT CHECK AMOUNT
220 IF B < 0 THEN GOTO 300 : REM TEST FOR OVERDRAFT
230 PRINT "YOUR NEW BALANCE IS", B
240 GOTO 20
300 PRINT "LAST CHECK CAUSES OVERDRAFT"
310 INPUT "DO YOU WISH TO CANCEL CHECK(Y/N)"; E$
320 IF E$="Y" THEN GOTO 400
330 PRINT "YOUR NEW BALANCE IS", B
340 GOTO 20
400 B=B + C: REM CANCEL LAST CHECK
410 GOTO 20
1000 END
```

Scan this program carefully to make sure you understand how each of the **INPUT** and **IF...THEN** statements are used. You should also use this program to obtain a feel for the dialog between you and your computer when **INPUT** statements are used.

Example 8. Write an Applesoft BASIC program which tests mastery in the addition of two-digit numbers. Let the user suggest the problems, and let the program keep score of the number correct out of ten.

Solution. Let us request that the program user suggest pairs of numbers via an **INPUT** statement. The sum will also be requested via an **INPUT** statement. An **IF...THEN** statement will be used to judge the correctness. The variable R will keep track of the number of correct responses. We will use a loop to repeat the process ten times.

```
10 N=1 : REM START COUNTER FOR 10 PROBLEMS
20 INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
30 INPUT "WHAT IS THEIR SUM"; C
40 IF A + B=C THEN GOTO 200
50 PRINT "SORRY. THE CORRECT ANSWER IS",A+B
60 GO TO 500 : REM GO TO THE NEXT PROBLEM
200 PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
210 R=R+1 : REM INCREASE SCORE BY 1
500 IF N < 10 THEN N=N+ 1: GOTO 20
600 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
700 PRINT "TO TRY AGAIN, TYPE RUN"
800 END
```

The power and utility of the IF. . .THEN statement can be further increased by using the logical connectors **AND** and **OR** to construct more complicated expressions for testing. For example:

```
10 IF S > 20000 OR YR >=20 THEN PRINT "SENIOR PERSON"
```

In this case, the message will be printed out if either the variable S exceeds 20000 or the variable YR is greater than or equal to 20 or both.

As another example,

```
20 IF D>=3 AND D < 9 THEN 400
```

In this example, the program will branch to line 400 only if the variable D is between 3 (inclusive) and 9 (exclusive). Note that in using the logical connectors there must be a full sentence on either side of the connector. We could not say

```
IF D>=3 AND<9 THEN ....
```

because the expression on the right of AND is not complete.

Infinite Loops and Control-C

As we have seen above, it is very convenient to be able to execute a loop without knowing in advance how many times the loop will be executed. With this convenience comes a danger. It is perfectly possible to create a loop which will be repeated indefinitely! For example, consider the following program:

```
10  N=1
20 PRINT N
30  N=N+1
40 GOTO 20
50 END
```

The variable N starts off at 1. We print it and then increase N by 1 (to 2), print it, increase N by 1 (to 3), print it, and so forth. This program will never stop of its own accord!

Such programs should clearly be avoided, but even experienced programmers occasionally create infinite loops. When this happens, there is no need to panic. There is a way of stopping the computer. Just press the keys **CONTROL** (**CTRL** on the Apple II Plus) and **C** simultaneously. (In the following we will refer to this combination of keys as the **"CONTROL-C"** combination.) This key sequence will interrupt the program currently in progress and return the computer to command mode. The computer is then ready to accept a command from the keyboard. Any program in RAM is undisturbed.

TEST YOUR UNDERSTANDING 4

Type the above program, RUN it, and stop it using the **CONTROL-C** combination.

Exercises (answers on page 311)

1. Write a BASIC program to check a charge customer's credit limit. Assume that information on the customer is kept in DATA statements. This includes the customer's name (Mary Jones), charge number (234-567), current balance due (150.00), and credit limit (1000). The program should READ in this information and then PRINT out a message asking for the new charges against the person and that charge number. When the new charges are INPUT, the computer will check to see if the new charges leave the balance due within the credit limit. It then outputs an appropriate message.
2. Write a BASIC program which will INPUT a number from the user, calculate its square and its square root, and PRINT out the answer.
3. Modify the arithmetic testing program of Example 8 so that the operation tested is for multiplication instead of addition.
4. Modify the arithmetic testing program of Example 8 so that it allows you to choose, at the beginning of each group of ten problems, from among these operations: addition, subtraction, or multiplication.
5. In many mathematical and scientific applications, it is often necessary to find the roots of a quadratic equation. A quadratic equation is one of the form

   $ax^2 + bx + c = 0$

   where a,b, and c are some constants, and the roots are those values of x which will make the equation true. There are two roots, and they can be found using the following equation:

   $$X = \frac{b \text{ +- square root of } (b^2 - 4ac)}{2a}$$

   The two roots come from the fact that one root is found by adding the square root term and the other by subtracting it. In most business-related problems the square root is positive. However, in some scientific problems we are required to take the root of a negative number. The square root of a negative number is called an imaginary number and is found by taking the square root of the absolute value (the value without the negative sign) and appending an "i" to denote that it is imaginary. For example, the square root of 4 is 2, but the square root of -4 is 2i.
   Write a BASIC program which will input the values of a,b, and c. Then the program should calculate $b^2 - 4ac$ to see if it is negative. If not, the computer should merely calculate the roots of the quadratic equation. If

the square root is negative, then the computer should calculate imaginary roots as:

$$x = \frac{b}{2a} +- \frac{\text{square root of [absolute value of } (b^2-4ac)\text{]} \text{ "i"}}{2a}$$

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, and 3

1: 100, 110, 120, 400

2: a.
```
10 IF A<B THEN PRINT A + B
20 END
```

b.
```
10 IF A^2 + D>=5000 THEN GOTO 300
20 GOTO 500
```

c.
```
10 IF N >=(I + K) THEN N=I + K
20 IF N < (I + K) THEN N=K
```

3: a.
```
10 INPUT "WHAT IS THE UNPAID BALANCE? ";B
20 IF B<=500 THEN S=.015 * B: GOTO 40
30 S=500 * .015 + (B-500) * .01
40 NW=B + S
50 PRINT "SERVICE CHARGE=";S
60 PRINT "NEW BALANCE=";NW
70 END
```

3.6 Application 3-1: Name The State Capitals

In this section we will develop a simple application which can be adapted to many learning situations. We have chosen to write it around the problem of learning the state capitals. In this problem the computer will give the name of a state and ask for the state capital. The person using the program will have three chances to correctly name the capital. (It must be spelled correctly, too!) If the user is correct on the first try, 10 points are added to the score; on the second try, only 6 points are gained; on the third, only 2 points. The computer runs through the states one by one, keeping track of the total score. The user has the opportunity to quit at any time.

All of the data is kept in DATA statements, so the computer reads in both the state and capital name before beginning the questioning. A state name of "NONE" is used to flag the end of the data.

These are the important variables used in the program:

S$	state name
C$	capital name
S	number of states handled thus far
G	number of guesses for this state thus far
G$	the user's guess
PT	points made on this state:

points are calculated as 14 - 4 times the number of guesses: if the user is correct on the first try, the points are 14 - 4=10; on the second try, 14 - 4*2=6; on the last try, 14 - 4*3=2 points.

TS total score for game

Here is the program.

```
5 HOME
10 READ S$: IF S$="NONE" THEN 150
20 READ C$
30 PRINT "WHAT IS THE CAPITAL OF ";S$
40 G=1: S=S+ 1
50 PRINT: PRINT: INPUT G$
60 IF G$=C$ THEN PRINT: PRINT: PRINT "CORRECT, ";:
     PT=PT - 4*G: TS=TS+PT: GOTO 100
80 IF G < 3 THEN PRINT "SORRY, TRY AGAIN ": G=G+1:
     GOTO 50
90 PRINT "SORRY, ";
100 PRINT "THE CAPITAL OF ";S$;" IS ";C$
110  PRINT: PRINT: PRINT "YOUR TOTAL SCORE IS ";TS;
     "AFTER ";S;" STATES"
120 INPUT "DO YOU WANT TO SEE MORE STATES? (Y/N)? ";A$
130 IF A$="Y" THEN HOME: GOTO 10
140 IF A$ <> "N" THEN 120
150  PRINT "THERE ARE NO MORE STATES": PRINT: PRINT:
          PRINT: "YOUR FINAL SCORE IS ";TS: PRINT
160 PRINT "HIGHEST POSSIBLE SCORE IS ";S*10: PRINT
170 PRINT: PRINT "IF YOU WISH TO PLAY AGAIN, TYPE RUN"
180 END
1000 DATA ILLINOIS,SPRINGFIELD
1010 DATA WASHINGTON, OLYMPIA
1020 DATA PENNSYLVANIA, HARRISBURG
.
.
.
9000 DATA NONE
```

Line 60 tests for the correct answer. If the answer is not correct, line 80 tests to see if the user has had three guesses yet. If the answer is wrong and there have been three guesses, the computer says "Sorry" and writes out the correct answer. Note that no points are added to the total score if the answer is wrong.

In Section 5.1 we will look at this application again. At that time we will make the selection of the state random so that the user does not memorize a sequence of answers.

Exercises

1. Write a program similar to the one in this chapter for some other subject. You may wish to use presidents of the United States (Who was the twenty-first president?) or sports trivia, for example.

3.7 More On Editing Lines

Suppose you discover a program line with an error in it. How can you correct it? Up to now, the only way was to retype the line. There is a much better way. The Apple II Computer allows you to **edit** lines. That is, you may add, delete, or change text in existing program lines.

The fundamental idea of editing on the Apple II is to recopy the line while making the needed corrections. The recopy and correction procedures require you to move the cursor by using certain key combinations. Let's begin by describing how to move the cursor.

The following keys move the cursor without affecting anything already on the screen.

I—move cursor one space up
J—move cursor one space to the left
K—move cursor one space right
M—move cursor one space down

Of course, these keys also generate the letters I, J, K, and M. To use them for moving the cursor, hit the key marked **ESC** (ESCAPE). You may return these keys to their normal functions by typing any key other than I, J, K, M, ESC, the REPT key on the Apple II Plus, or the arrow keys on the Apple IIe.

On the Apple II Plus you may find it helpful in moving the cursor to use the **REPT** key in conjunction with the above keys. If you hold the **REPT** key down while pressing I, for instance, the cursor will execute a continuous move up. You may stop the cursor by lifting your fingers from the keys.

On the Apple IIe it is only necessary to continue pressing the key that you want. The Apple IIe features automatic key repeat. On this model you can also move the cursor in editing mode with the four arrow keys. Again, you need to press ESCape first to get you into editing mode. The upward arrow can be used to move up one line; the downward arrow will move you down one line.

On the Apple IIe the left pointing arrow will move the cursor to the left and the right pointing arrow will move the cursor to the right. Press ESCape and then press the right arrow. While the cursor moves to the right, it does not move one space at a time. While you are in editing mode, the left and right arrows will move the cursor eight spaces in the selected direction. This provides a quick means of moving about the line. The left arrow cannot be used to delete characters while you are in editing mode.

To be sure you understand how these keys work, sit in front of your Apple II Computer and clear the screen by typing in **HOME**. Now the cursor is on the upper left corner of the screen. Hit the ESCape (ESC) key. Now, if you have an Apple IIe, hold the M key or the down arrow key down. You should see the cursor continue down the screen as long as you press the key. If you have an Apple II Plus, you will have to hold the M key down and then hold the Repeat (REPT) key also.

Repeat this same operation trying the I, J, and K keys (or the other arrow keys on the Apple IIe) to see how they cause the cursor to move. When you

wish to go back to typing the letters instead of moving the cursor, just hit any key except I, J, K, L, and ESC, the REPT key on the Apple II Plus, or the arrow keys on the Apple IIe. It is often convenient to use the space bar for this. When you feel comfortable about moving the cursor this way, you should continue with the editing process described in the following paragraphs.

On the Apple II Computer, the editing process consists of five steps:

1. Display the line on the screen. You can display an individual line by using a variation of the **LIST** command which we discussed earlier. All you need to do is type in

   ```
   LIST <line number>
   ```

   where line number is the number of the line you want displayed.
2. Position the cursor at the first character of the line number by hitting the ESC key and then using the I, J, K, and M keys, or the arrow keys on the Apple IIe, to move the cursor. When properly positioned, hit the space bar to get out of cursor movement mode.
3. "Copy" the line by using the right arrow key to move the cursor over each character of the line.
4. Add, change, or delete characters as they arise in the "copy" process.
5. Send the line to the computer via the **RETURN** key.

The best way to understand the editing process is to work through several examples. Follow these examples by typing them out on your keyboard.

Suppose you have typed the following program lines:

```
10 PRIMT X,Y,Z
20 IF A=5 THN 50: GOTO 30
 ]
```

The third line indicates the cursor position. We immediately see that there are two spelling errors: PRIMT and THN. (If the computer had any common sense, it would have known what you meant.) In addition, suppose we wish to change X, Y, and Z in the first line to read: A, B, X, Y, Z. Finally, suppose we wish to delete the :GOTO 30 on the second line. Let's use the editing process to correct these statements, making one change at a time.

We can display the line we wish to edit by typing in **LIST <line number>** (Step 1). Next, we move the cursor to the first digit of the line number using the cursor movement techniques described above (Step 2). Now we wish to "copy" those characters which are correct. The copy operation is performed using the right pointing arrow to move the cursor across them (Step 3). Only the right pointing arrow can accomplish the copy operation!! We move the cursor until it rests on the M in PRIMT. To correct the M we type N (Step 4). The display now looks like this:

```
10 PRINT X,Y,Z
```

The first error has now been corrected.

We now continue to copy the line with the right pointing arrow key to move the cursor. Before adding the additional items to the variable list, let's

move the cursor (using the right arrow) to the end of the line and hit the **RETURN** key (Step 5). If you list the line now, you should see that it reads

```
10 PRINT X,Y,Z
```

Now let's add the additional variables A and B before the X. Position the cursor at the start of the line number using the ESC key sequence. Then use the right arrow to copy the correct characters and position the cursor on the X. We must insert the characters **A, B,** before the X. Here is the display:

```
10 PRINT X,Y,Z
```

To insert text at the cursor position, we type ESC I (or ESC and then the upward arrow on the IIe) to move the cursor up one line. Then hit another key to get out of the cursor movement mode (for example, hit the space bar). We then type the material to be inserted. Here is the display:

```
          A,B,
10 PRINT X,Y,Z
```

We now type **ESC M** (or ESC and the down arrow on the IIe) to bring the cursor down one line and type J four times to move the cursor back to the X. Now exit the cursor movement mode by typing another key (space bar, perhaps) and resume the copy operation using the right arrow key. After you pass over the last character of the line, hit **RETURN**. This finishes the corrections on the first line.

Using the ESC key to add characters requires some practice. Remember that you must hit a key to get out of the cursor movement mode before typing what you want to insert.

We are now ready to edit the second line. Type **LIST 20** followed by **RETURN**. This will display line 20.

```
LIST 20

20 IF A=5 THN 50 :GOTO 30
]
```

We move the cursor to the first character of line 20 using the **ESC** sequence. Then we begin the copy operation as we did in correcting line 10, by using the ESC key to move the cursor to the first character of the line number. We correct the misspelling of THEN by moving the cursor to the N (14 spaces to the right). We then hit the **ESC-I** (or ESC-up arrow on the IIe) sequence to move the cursor up and hit the space bar to get the computer out of cursor movement mode. Type in the letter E, then use the **ESC** M (or ESC-down arrow on IIe) sequence to move down. Then hit J to move left one space. Get out of cursor movement mode and use the right arrow to "copy" the remaining letters.

The final correction is to delete :GOTO 30. This is done by positioning the cursor on the 0 in 30 and hitting the backspace key (←). As we backspace over a letter, it is erased. Finally, we hit the **RETURN** key to enter the corrected line. Here is an easier way of deleting the :GOTO 30. Position the cursor at the first character to be deleted (the colon) and hit the **RETURN** key.

The above example illustrates the various editing features of the Apple II Computer. We may use the editing keys in the same way to alter any line on the screen.

IMPORTANT NOTE 1: Editing changes occur only in the copy of the program in RAM. In order for changes to be reflected in copies of the program on cassette or diskette, it is necessary to *save the edited copy of the program.*

IMPORTANT NOTE 2: You may have noticed that when the computer displays the program you have written as a result of a LIST command, the program does not use up all 40 columns. This can cause problems when trying to edit if the computer puts something which is in quotes in a PRINT statement on two lines. To prevent this from causing problems, type in

```
POKE 33,33
```

before beginning the editing process. When finished, type in

```
TEXT
```

We will explain these statements in Chapter Eight.

Exercises

What keystrokes accomplish the following editing functions?

1. Move the cursor four spaces to the right.
2. Delete the fourth letter to the right of the cursor.
3. Insert the characters 538 at the current cursor position.
4. Delete the portion of the line to the right of the cursor position.
5. Move the cursor up eight spaces.
6. Move the cursor to the right three spaces.
7. List the current version of the line.
8. Change 0 to a 1 at the current cursor position.
9. Delete the letter "a" eight spaces to the left of the current cursor position.
10. Cancel all changes in the current line.

Use the line editor to make the indicated changes in the following program line. The exercises are to be done in order.

```
300 IFF M=11 AMD D < 5 THEN, X=M^2 - 5
```

11. Delete the ,
12. Correct the misspelling of the word IF.
13. Change M^2 - 5 to M^3 - 2 .
14. Change 5 to 25 .
15. Add the following characters to the end of the line : Y=M + 1 .

3.8 Some BASIC Functions

In performing computations it is often necessary to use a wide variety of different functions. The Apple II Computer has a wide range of these functions "built-in." In this section we will describe these functions and their use. Some of these functions may not be of any use to you because they are of value in scientific calculations. You may choose to skip these functions when we describe them.

All "built-in" functions in Applesoft BASIC work in a similar fashion. Each function is identified by a sequence of letters (**SQR** for square root, **LOG** for natural logarithm, and so forth).

To evaluate a function at a number X, we write X in parentheses after the function name. For example, the square root of X is written SQR(X). The program will use the current value of the variable X and will evaluate the square root of that value. For example, if X is currently 2, then the computer will calculate SQR(2).

Instead of the variable X, we may use any type of numeric variable integer or real. We may also use a numerical constant of any type. For example, LOG(.435678889658595) asks for the natural logarithm of a real numerical constant. Note that with only a few exceptions (see below), all Applesoft BASIC functions return a real result, accurate to nine digits. For example, the above value of the log function will be computed as

```
LOG(.435678889658595)=-.830849799
```

Applesoft BASIC lets you evaluate a function of any expression. Consider the expression X^2 + Y^2 -3*X. It is perfectly acceptable to call for calculations such as

```
SQR(X^2+Y^2 - 3*X)
```

The computer will first evaluate the expression X^2 + Y^2 - 3*X using the current values of the variables X and Y. If X=1 and Y=4, then X^2 + Y^2 - 3*X=1^2 + 4^2 - 3*1=14. So the above sine function will be evaluated as SQR(14)=3.74165739.

Powers

In introducing functions in this chapter, we used the square root function as an example. Let's look at this function again. This function, denoted as SQR(X), will accept as input an integer or a real constant and will output a real constant. For example, the instruction

```
10 Y=SQR(2)
```

will set Y equal to 1.41421356.

Actually, the exponentiation procedure which we learned in Section 3.3 will work equally well for fractional and decimal exponents and therefore provides an alternative method for extracting square roots. Here is how to

use it. Taking the square root of a number corresponds to raising the number to the 1/2 power. We may calculate the square root of X as

```
X^(1/2)
```

The square root function SQR(X) operates with greater speed and is therefore preferred, but the alternate method is more flexible. For instance, we may extract the cube root of X as

```
X^(1/3)
```

or we may raise X to the 5.389 power as follows:

```
X^5.389
```

Greatest Integer, Absolute Value, and Related Functions

Here are several extremely helpful functions. The greatest integer less than or equal to X is denoted **INT(X)**. For example, the largest integer less than or equal to 5.46789 is 5, so

```
INT(5.46789)=5
```

Similarly, the largest integer less than or equal to −3.4 is −4. This is because on the number line −4 is the first integer to the left of −3.4. Therefore:

```
INT(-3.4)=-4
```

Note that the **INT** function throws away the decimal part of a positive number although this description does not apply to negative numbers.

The **INT** function can be very useful in rounding numbers and controlling the number of decimal places which are printed out. For example, type in and run the following program:

```
10 X=345.678
20 PRINT X
```

The output of this program will be

```
345.678
```

This is what we expected, but suppose this amount represented dollars and cents so that we would want to round the number to two decimal places. In this example we would want to print out 345.68.

To do this we can take the value of X and multiply it by 100. This has the effect of moving the decimal point two places to the right and yields

```
34567.8
```

If we now add .5 to this new number, the result of the addition will make our value equal to

```
34568.3
```

As you can see, the value to the left of the decimal place is beginning to resemble the desired output. At this point we can use the **INT** function to chop off the decimal places, leaving

```
34568
```

Dividing this value by 100 will now move the decimal point two places to the left and give us the desired rounded value. The entire procedure is

```
10 INPUT X
20 Y=INT(100*X + .5)/100
30 PRINT Y
```

The original assignment statement in line 10 was changed to an INPUT statement so that you can try running the program with different values of X. As you will see, the program will not provide two decimal places if trailing zeroes are required (try using 345.6 and 12 as values of X). Nor will it work if the number is so large it requires exponential notation. We will refine this procedure later in the book so it can handle all possibilities.

The absolute value of X is denoted ABS(X). Recall that the absolute value of X is X itself if X is positive or 0 and equals −X if X is negative. For example:

```
ABS(9.23)=9.23
ABS(0)=0
ABS(-4.1)=4.1
```

Just as the absolute value of X "removes the sign" of X, the function SGN(X) throws away the number and leaves only a −1 if the value of X was negative and a 1 if the value was positive. The exception is a value of zero. For example:

```
SGN(3.4)=+1
SGN(-5.62)=-1
SGN(0)=0.
```

Example 1. Write an Applesoft BASIC program to graph the function y = ABS(x) for values of x from −9 to +9 inclusive.

Solution. The first step in the solution is deciding how the output should look. Because we are still somewhat limited in how we move things around on the screen, let's print the graph sideways. It will look like the picture in Figure 3-2.

Notice that we have used exclamation points for the x-axis and dashes for the y-axis. The x-axis is labeled "X", and the points are plotted as the letter "O". We have numbered the x-axis so that the numbers align. This means we will have to tab one additional column when we come to the positive numbers.

Let's assume we want to put the numbers in column 5 and the x-axis in column 10. We will use a variable, I, which we will change from 0 to 1 when we reach the positive numbers to give us the correct tab location. Here is the program:

```
10 X=-9: I=0
20 Y=ABS(X)
30 IF X=0   THEN   PRINT   "X";    TAB(6);
    "0";TAB(10);"0-------------": I=1: GOTO 50
40 PRINT TAB(5 + I); X; TAB(10); "!"; TAB(10 + Y); "0"
50 X=X + 1
60 IF X < 10 THEN GOTO 20
70 END
```

```
     -9 !          0
     -8 !         0
     -7 !        0
     -6 !       0
     -5 !      0
     -4 !     0
     -3 !    0
     -2 !   0
     -1 !  0
X     0 0--------
      1 !  0
      2 !   0
      3 !    0
      4 !     0
      5 !      0
      6 !       0
      7 !        0
      8 !         0
      9 !          0
```

Figure 3-2: Output from absolute value plotting program.

Generating Chance Events

One of the most interesting features of your computer is its ability to allow us to "observe" events with "random" outcomes. For example, you may instruct the computer to "throw a pair of dice" and produce a random pair of integers between 1 and 6. You may instruct the computer to "pick a card at random from a deck of 52 cards." You may also program the computer to choose a two-digit number "at random."

The source of all such random choices is the **random number generator** which is part of Applesoft BASIC. Let us begin by explaining what the random number generator is and how to access it.

You may generate random numbers using the Applesoft BASIC function **RND(1)**. To explain how this function works, let us consider the following program:

```
10 X=1
20 PRINT RND(1)
30 IF X<500 THEN X=X+1: GOTO 20
40 END
```

This program consists of a loop which prints 500 numbers, each called **RND(1)**. Each of these numbers lies between 0.0000000000 (inclusive) and 1.000000000 (exclusive). Each time **RND(1)** is called (as in line 20 above) the computer makes a "random" choice from among the numbers in the indicated range. This is the number that is printed.

To get a better idea of what we are talking about, you should generate some random numbers using a program like the one above. Unless you have a printer, 500 numbers will be too many for you to look at in one viewing. You should print two random numbers on one line (one per print zone) and limit yourself to 24 displayed lines at one time. Here is a partial printout of such a program.

```
.2451213213     .3050034958
.9845468610     .9011598475
.8966099342     .6602124859
.5839383519     .4481638341
.1371193105     .2265442384
```

What makes these numbers "random" is that the procedure the computer uses to select them is "unbiased," with all numbers having an equal likelihood of selection. If you generate a large collection of random numbers, numbers between 0 and .1 will comprise approximately 10 percent of those chosen, those between .5 and 1.0 will comprise 50 percent of those chosen, and so forth. In some sense, the random number generator provides a uniform sample of the numbers between 0 and 1.

TEST YOUR UNDERSTANDING 1 (answer on page 84)

Assume that RND(1) is used to generate 1000 numbers. Approximately how many of these numbers would you expect to lie between .6 and .9?

The function **RND(1)** generates random numbers lying between 0 and 1, but in many applications we will require randomly chosen **integers** lying in a different range.

Suppose that we wish to generate random integers chosen from among 1,2,3,4,5,6. Let us multiply **RND(1)** by 6, to obtain **6*RND(1)**. This is a random number between 0.0000000000 and 5.999999999. Next, let us add 1 to this number. Then 6*RND(1)+1 is a random number between 1.000000000 and 6.999999999. To obtain integers from among 1,2,3,4,5,6, we must "chop off" the decimal portion of the number 6*RND(1)+1. To do this we use the **INT** function. If X is any number, then INT(X) is the largest integer less than or equal to X. For example

```
INT(5.23)=5, INT(7.99)=7, INT(100.001)=100
```

Be careful in using INT with negative X. The definition we gave is correct, but unless you think things through, it is easy to make an error. For example

```
INT(-7.4)=-8
```

since the largest integer less than or equal to -7.4 is equal to -8. (Draw -7.4 and -8 on a number line to see the point!)

Let us get back to our random numbers. To chop off the decimal portion of 6*RND(1)+1, we compute INT(6*RND(1)+1). This last expression is a random number from among 1,2,3,4,5,6. Similarly, the expression

```
INT(100*RND(1)+1)
```

may be used to generate random numbers from among the integers 1,2,3, . . . ,100.

TEST YOUR UNDERSTANDING 2 (answer on page 84)

Generate random integers from 0 to 1. This is the computer analog of flipping a coin: 0 = heads, 1 = tails. Run this program to generate 50 coin tosses. Count the number of heads and tails which occur.

Example 2. Write a program which turns the computer into a pair of dice. Your program should report the number rolled on each as well as the total.

Solution. We will hold the value of die #1 in the variable D1 and the value of die #2 in variable D2. The program will compute values for D1 and D2, print out the values and the total D1 + D2.

```
10 HOME
20 D1=INT(6*RND(1)+1)
30 D2=INT(6*RND(1)+1)
40 PRINT "LADIES AND GENTLEMEN, BETS PLEASE!"
50 INPUT "ARE ALL BETS DOWN(Y/N)"; A$
60 IF A$="N" THEN 40
100 PRINT "THE ROLL IS",D1;SPC(3);D2
110 PRINT "THE WINNING TOTAL IS " ; D1+D2
120 INPUT "PLAY AGAIN(Y/N)"; B$
130 IF B$="Y" THEN 10
200 PRINT "THE CASINO IS CLOSING. SORRY!"
210 END
```

Note the use of computer-generated conversation on the screen and how the program uses lines 120-130 to allow the player to control how many times the game will be played.

TEST YOUR UNDERSTANDING 3 (answer on page 84)

Write a program which flips a "biased coin." Let it report "heads" one-fourth of the time and tails three-fourths of the time.

Trigonometric Functions

The Apple II Computer has the following trigonometric functions available:

SIN(X) = the sine of the angle X
COS(X) = the cosine of the angle X
TAN(X) = the tangent of the angle X

Here the angle X is expressed in radian measure. In this measurement system, 360 degrees equals 2 radians. One degree equals .017453 radians and one radian equals 57.29578 degrees. If you want to calculate trigonometric functions with the angle X expressed in degrees, use these functions:

```
SIN(.017453*X)
COS(.017453*X)
TAN(.017453*X)
```

The three other trigonometric functions, SEC(X), CSC(X), and COT(X), may be computed from the formulas

SEC(X) = 1/COS(X)
CSC(X) = 1/SIN(X)
COT(X) = COS(X)/SIN(X)

Here, as before, the angle X is in radians. To compute these trigonometric functions with the angle in degrees, replace X by

```
.017453*X
```

Applesoft BASIC has only one of the inverse trigonometric functions, namely the arctangent, denoted ATN(X). This function returns the angle whose tangent is X. The angle returned is expressed in radians. To compute the arctangent with the angle expressed in degrees, use the function

```
57.29578*ATN(X)
```

TEST YOUR UNDERSTANDING 4 (answer on page 84)

Write a program which calculates sin 45°, cos 45°and tan 45°

Logarithmic and Exponential Functions

In many scientific calculations some constants appear quite frequently. One of these is a constant which is denoted as *e* that is approximately equal to 2.17828183. To help in simplifying some of these scientific calculations, Applesoft BASIC allows you to compute e^X using the exponential function

```
EXP(X)
```

Furthermore, you may compute the natural logarithm of X via the function

```
LOG(X)
```

You may calculate logarithms to base b using the formula

$LOG_b(X) = LOG(X)/LOG(b)$

Example 3. Prepare a table of values of the natural logarithm function for values X = .01, .10, 1.00, 10.00, 100.00

Solution. Here is the desired program. Note that we have prepared our table in two columns with a heading over each column.

```
10 PRINT "X", "LOG(X)"
20 J=.01
30 PRINT J, LOG(J)
40 J=J * 10
50 IF J>100.0 THEN 100
60 GOTO 30
100 END
```

TEST YOUR UNDERSTANDING 5 (answers on page 84)

Write a program which evaluates the function
$f(x) = \sin x/(\log x + e^x)$ for $x = .45$ and $x = .7$.

Example 4. Carbon dating is a technique for calculating the age of ancient artifacts by measuring the amount of radioactive carbon-14 remaining in the artifact and comparing it with the amount that would be present if the artifact were manufactured today. If r denotes the proportion of carbon-14 remaining, then the age A of the object is calculated from the formula

```
A=-(1/.00012)*LOG(r)
```

Suppose that a papyrus scroll contains 47% of the carbon-14 found in a newly manufactured piece of papyrus. Calculate the age of the scroll.

Solution. Here r = .47. So we use the above formula:

```
10 R=.47
20 A=-(1/.00012)*LOG(R)
30 PRINT "THE AGE OF THE PAPYRUS IS", A, "YEARS"
40 END
```

Exercises (answers on page 312)

1. Write a program to calculate the fractional part of x. (The fractional part of x is the portion of x which lies to the right of the decimal point.)
2. Write Applesoft BASIC programs which generate random numbers of the following sorts:

   a. Numbers from 0 to 100.
   b. Integers from 1 to 50.

3. Write BASIC statements to calculate the following quantities:
   a. $e^{1.54}$ b. $e^{-2.376}$

c. log 58 d. log .0000975
e. sin 3.7 f. cos 45°
g. arctan 1 h. tan .682

4. Write a BASIC program to create a table of values of the exponential function exp(x) for x = −1.0, −0.9, . . . , 0 , .1, . . . , 1.0.
5. Write a BASIC program to evaluate the function

$$3x^{1/4} \log(5x) + e^{1.8x} \tan x$$

for x = 1.7, 3.1, 5.9, 7.8, 8.4, and 10.1. Use READ . . . DATA statements to obtain the x values.
6. Write an Applesoft BASIC program to graph the function y = sin x for x from 0 to 25. Use an interval of .04 on the x axis.
7. Modify the program of Exercise 4, Section 3.5, so that the computer randomly generates the numbers to be used in the arithmetic problems.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, 3, 4, and 5

1: 100% of the numbers lie between 0 and 1 and they are evenly distributed over this interval. This means about 30% of the 1000 numbers generated, or about 300, will lie between .6 and .9.

2:
```
10 I=1:H=0:T=0
20 C=RND(1)
30 IF C<.5 THEN H=H+1
40 IF C>=.5 THEN T=T+1
50 I=I + 1
60 IF I<50 THEN GOTO 20
70 PRINT "NUMBER OF HEADS=";H
80 PRINT "NUMBER OF TAILS=";T
90 END
```

3:
```
10 I=1:H=0:T=0
20 C=RND(1)
30 IF C<.25 THEN H=H+1
40 IF C>=.25 THEN T=T+1
50 I=I + 1
60 IF I<50 THEN GOTO 20
70 PRINT "NUMBER OF HEADS=";H
80 PRINT "NUMBER OF TAILS=";T
90 END
```

4:
```
10 A=.17453
20 PRINT SIN(45*A), COS(45*A), TAN(45*A)
30 END
```

5:
```
10 DATA .45,.7
20 J=1 :REM SET COUNTER FOR TWO PIECES OF DATA
30 READ A
40 PRINT SIN(A)/(LOG(A)+EXP(A))
50 IF J <2 THEN J=J+1:GOTO 30
60 END
```

4
Structuring a Solution

So far, the programs which we have been looking at have been fairly simple. As we learn more about the capabilities of BASIC, we will want to write programs which are more lengthy and complex.

You might be wondering how it is possible to plan and execute such programs. The key is to divide the solution into a set of smaller problems so each smaller problem can be worked on individually. This may sound very simple and logical, but you will find that the discipline required to structure the solution before starting to write any BASIC code is sometimes difficult to adhere to.

In this chapter we will look at two commonly used techniques for outlining the solution. In practice there is no one correct way to do this. The one that works best for you is the one which you should use, whether it is a detailed verbal description or a picture of the solution process.

There is only one important rule to remember:

Don't start writing any BASIC statements until the solution procedure is structured.

Before we proceed to discuss ways to structure a solution procedure, let's look at the steps involved in actually developing a program to solve a problem.

Step 1: Understand the problem.
This may seem a fairly simple thing to do, yet it is often the source of logic errors in the program. One of the best ways to make sure you understand the problem is to make up some simple numeric example. If you can solve the problem with some simple data, you probably understand what is required. In the feet to meters problem, we could take 3 feet and convert it to meters by multiplying by .3048 to obtain .9144 meters as an answer.

Step 2: Develop a solution procedure.
Once you have a good grasp of the problem, you can begin to decide how to approach the solution. Sometimes there aren't many alternate ways; in the feet to meters problem or the problem to add two numbers together, the approach was straightforward. As the problems become more com-

plex, there may be many ways to approach the solution. Using some simple numeric examples will help you explore the possibilities.

The general approach is to take the problem and divide it into major blocks, each of which can be developed separately. These major logical blocks are then subdivided until we have a solution procedure which can be implemented.

Step 3: Translate the solution procedure into BASIC.
If the solution procedure is divided into workable blocks, the program can be developed the same way, with the pieces of the program being written and tested separately. This will be discussed in greater detail in Chapter Six. The solution procedure should be developed in sufficient detail so that translation into BASIC is relatively straightforward.

Our rationale for discussing the development of the solution process and the development of the program itself separately is to emphasize that these are indeed two separate steps. Developing a good program requires you to develop a good solution procedure first.

Step 4: Enter the program into the computer and execute it.
Once we have the program written, we can enter it into the computer by typing it on the keyboard. This is a separate step from actually developing the program. The program should be written on paper first. Failure to do so will usually result in a lot of wasted time and effort and more hours in front of the computer than is necessary.

Step 5: Examine the results and revise as necessary.
Just because the program runs and produces output does not mean it is correct. A necessary part of the process is examining the output to see if it makes sense. If you began by using some simple numeric example to be sure you understood the problem, then this same problem can be used to see if the correct answer is obtained. The process of checking and correcting your program will be further discussed in Chapter Five.

Even though we have listed these steps as sequential, the process of developing a good program generally involves going back to some previous steps several times. Sometimes this is because you encounter unforeseen difficulties in implementing your program in BASIC and sometimes because you simply did not consider some important contingency. If you keep these steps in mind, you should greatly simplify the process of developing a program.

4.1 A Step-wise Approach to Structuring a Solution

The first approach we will examine is to divide the procedure into major blocks. Each of these blocks can be subdivided as necessary until we have a sufficient degree of detail.

As an example, let's use the feet to meters problem discussed in Chapter Two. In that problem we wanted to change any number of feet to the correct,

equivalent number of meters. To understand the problem we took 3 feet and multiplied it by the conversion factor .3048 to obtain the equivalent number of meters.

This is a straightforward problem. The major steps are

```
get the number of feet
multiply the feet by the conversion factor .3048
write out the result
```

These are the three major steps in the process. To make this a little more systematic, we can give each block a name.

```
initialize: get the number of feet
convert: multiply feet by .3048
output: write out the result
```

Each of these three blocks is very simple and straightforward. The logic proceeds sequentially.

The BASIC program can easily be developed from this solution process. To see how the program steps correspond to the steps in the solution procedure, you may wish to review Figure 2-1.

Let's take another example. Assume we wish to find the maximum value from among three numbers. First be sure you understand the problem.

```
If the numbers are 23, 45, and 54, the answer is 54
If the numbers are 34, 12, and 34 the answer is 34
If the numbers are 12, 12, and 12 the answer is 12
```

You are only looking for the maximum value, not which of the three numbers have that value.

Now let's try to structure a solution. The steps are

```
initialize: get the numbers (call them A, B, and C)
find maximum: get the largest value
output: write out the answer
```

In this example we also have three major blocks, but the second step, **find maximum**, requires more detail in order to implement. Let's try to figure out how to find the maximum.

We can compare the first two numbers to see which is largest and then compare that result to the third number. The procedure is

```
find maximum:
   find maximum A,B: compare A to B, call the largest
        value MX
   find maximum MX,C: if C is larger than MX, change the
        value of MX to equal C
```

The **find maximum** block is now sufficiently subdivided so that the BASIC program can be written directly from the procedure.

The complete specification of the solution procedure is:

```
initialize: get the numbers (call them A, B, and C)
find maximum: get the largest value
```

```
    find maximum A,B: compare A to B, call the largest
         value MX
    find maximum MX,C: if C is larger than MX, change the
         value of MX to equal C
output: write out the answer
```

TEST YOUR UNDERSTANDING 1 (answer on page 89)

Write a BASIC program to find the largest value from among three numbers based on the solution procedure developed above.

In each of these examples the logic flowed sequentially from top to bottom. In many important cases we will encounter a loop. Let's look at the problem of Examples 1 and 2 (page 60) of Chapter Three. The problem was:

A firm keeps records on its employees. The data kept includes the employee's name, gross pay for each pay period, and the tax rate for that individual. Write a program which will read in the data for each employee one after the other, calculate the net pay, and write out the employee's name and net pay. The major blocks for this problem are:

```
loop until no more employees
   initialize: get employee name, pay, tax rate
   net pay: multiply gross by 1 minus the tax rate
   output: write out result
endloop
```

While the three blocks look familiar, we have now enclosed them in a loop because we will want to repeat this process for each employee. The indentation helps us to see the logical structure of the problem.

The program for this problem can now easily be written. Let us denote the employee's name as N$, the gross pay as GR, the tax rate as TX, and the net pay as NT. Employee data will be kept in DATA statements with the last DATA statement containing sentinal data.

```
5 PRINT " NAME";TAB(25);"NET PAY"
10 READ N$,GR,TX
15 IF N$="NOMORE" THEN END
20 NT=GR * (1 - TX)
30 PRINT N$;TAB(25);NT
40 GOTO 10
100 DATA MARY JONES,345.00,.20
110 DATA PETER PAUL,310.00,.15
900 DATA NOMORE,0,0
```

The **initialize** section does not specify how this initialization process is to be done in the program. Further, the program will generally include statements which make the program easier to use, usually by adding additional explanation by way of PRINT statements or REMarks. In this case line 5 provided headings for the output.

The approach outlined in this section is the approach we will generally use to structure solution procedures in this book. Remember that there is no hard and fast rule about how it should be done or what specific words should be used and that only a systematic approach to outlining the solution procedure should be adopted. You can use any method you find useful.

In the next section we will explore another method which many people find helpful in setting up their solution procedures.

ANSWER TO TEST YOUR UNDERSTANDING 1

```
10 PRINT "FIND THE LARGEST VALUE FROM THREE NUMBERS"
20 INPUT "TYPE IN THE THREE NUMBERS ";A,B,C
30    MX=A: IF B>MX THEN MX=B
40    IF C>MX THEN MX=C
50 PRINT "THE LARGEST VALUE IS ";MX
60 END
```

Line 20 corresponds to the *initialization* block. Lines 30 and 40 are from the *find maximum* section. We have deliberately indented them to show that they correspond to the subdivisions of this block. This is not necessary in your program. Line 30 sets the maximum value equal to the value of A then changes it if necessary. Line 40 compares the value of C to this current maximum value and changes the maximum if C is larger. Finally, line 50 is the *output* section.

4.2 Flowcharting

The old saying "A picture is worth a thousand words" is true for computer programming. In designing a program, especially a long one, it is helpful to draw a picture of the program instructions of the program and their interrelationships. Such a picture is called a *flowchart.*

A flowchart is a series of boxes connected by arrows. Within each box is a series of one or more computer instructions. The arrows indicate the logical flow of the instructions.

For example, let's use the feet to meters problem again. The flowchart for this problem is shown in Figure 4-1. The arrows indicate the sequence of operations. Note the similarity between the block structure we used in Section 4.1 and this more pictorial presentation of the solution process. Either one would easily lead to the correct program.

There are many flowcharting rules used by professional programmers. Different shapes of boxes represent certain programming operations. For our purposes this kind of detail is unnecessary. We will adopt the very simple rule that all boxes are rectangular except for decision boxes. Decision boxes are diamond-shaped.

Let's use the flowcharting approach on the problem of finding the largest of three numbers. This is shown in Figure 4-2 and illustrates the use of decision boxes. Each diamond-shaped box has two arrows leading from it, indi-

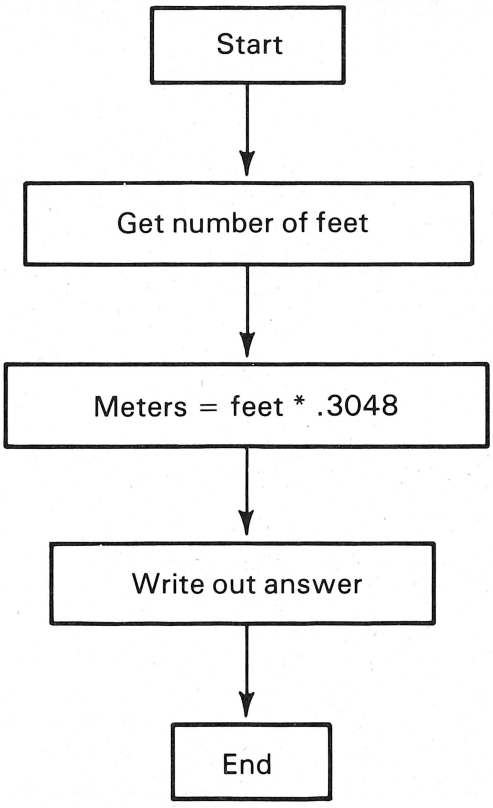


Figure 4-1. FEET-TO-METERS flowchart.

cating the two possible answers to the question posed and the sequence of actions required for each answer.

Now let's consider the net pay problem which we discussed in Section 4.1. This problem requires a looping process, so we will see one method of depicting loops in a flowchart. This is shown in Figure 4-3. In this case the arrow indicates a return to a previous step.

You will find flowcharting helpful in thinking out the necessary steps of a program and in helping others to understand your logic. As you practice flowcharting, you will develop your own style and conventions. Remember that any approach is fine as long as you try to structure the solution *before* writing the program.

Exercises

Develop structured solution procedures for the following problems. Use the method most comfortable for you.

1. Read a file of employee information including the name, salary, and number of years with the company. Prepare a list of all employees

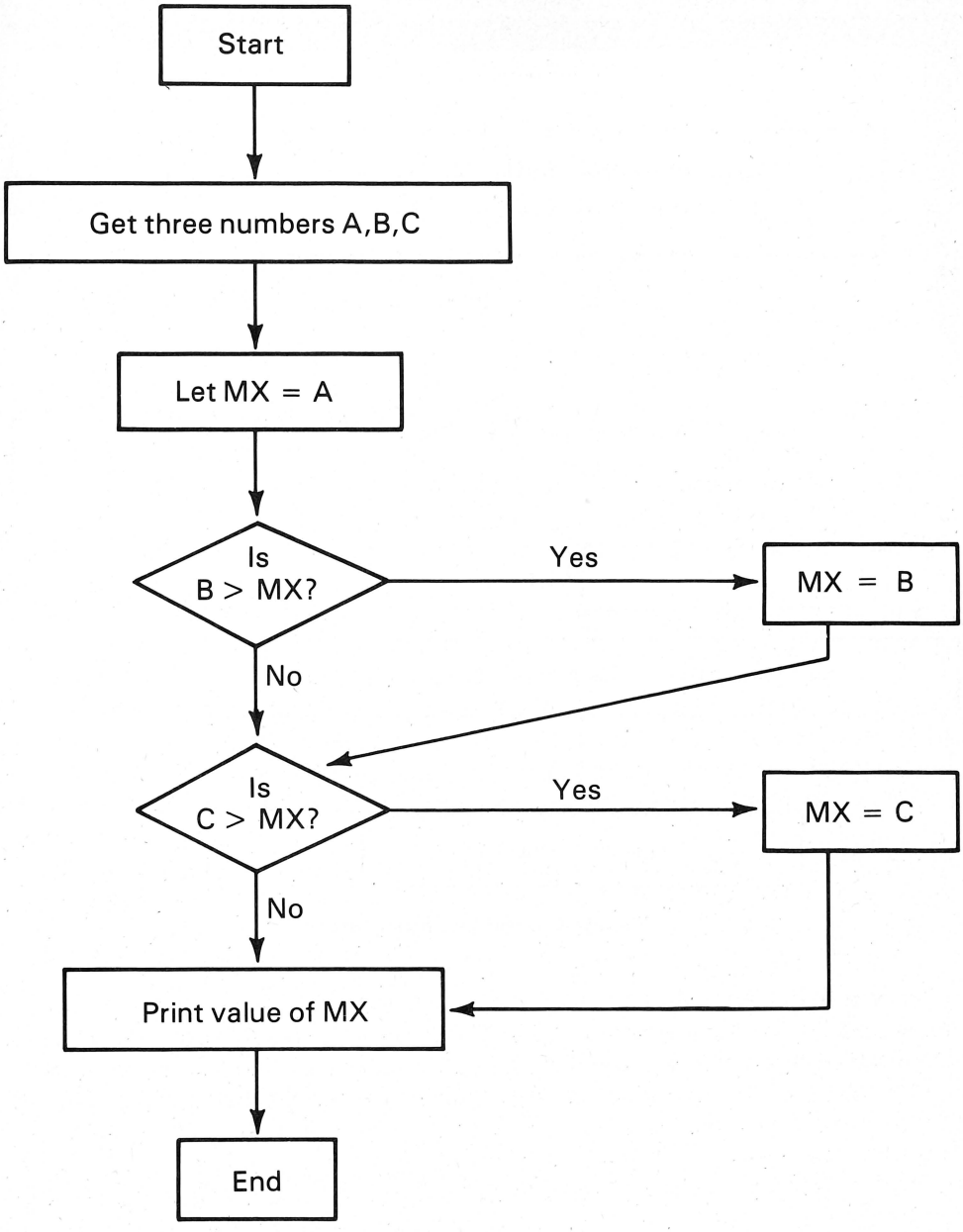


Figure 4-2. Maximum value flowchart.

who either have 20 years of service or who earn a salary of $30,000 or more. Remember to check for the end of the data.

2. Write a solution procedure to calculate the squares of the integers from 1 to 20.

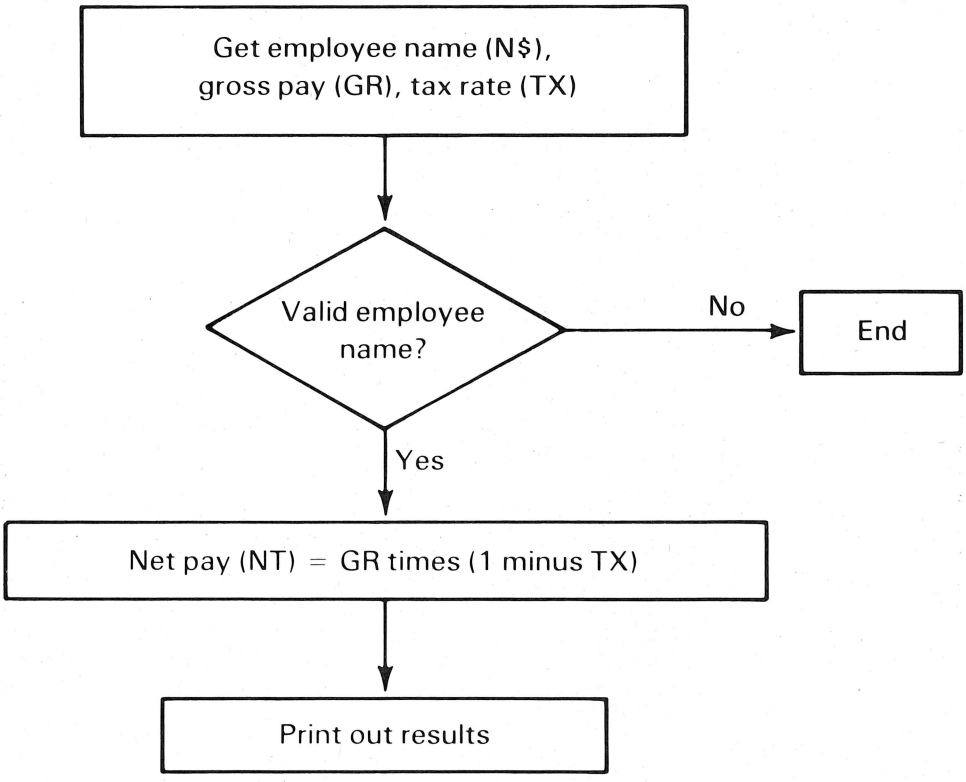


Figure 4-3. Net pay flowchart.

5
More About BASIC

5.1 Doing Repetitive Operations

In Chapter Three we discussed some techniques for getting the computer to loop and repeat a sequence of operations. In those cases we used the GOTO statement to loop, and we tested for some counter or flag data to stop the looping process.

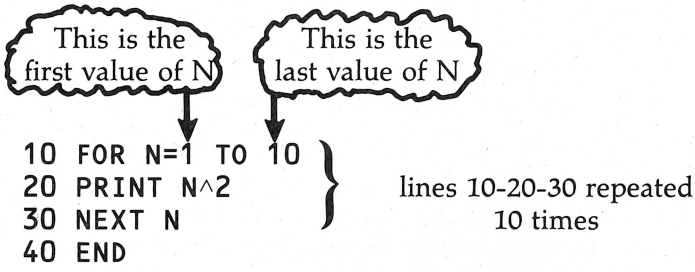
In cases where we know how many times we want to repeat the operations, there is an easier way to effect a loop. The pair of BASIC statements needed to do this is referred to as a **FOR . . . NEXT** loop.

Suppose we wish to solve 50 similar multiplication problems, or 500, or even 5000. Typing the problems one at a time would not be practical. If we can describe to the computer the entire class of problems we want solved, then we can instruct the computer to solve them using the **FOR . . . NEXT** structure.

Let us consider a concrete problem. Suppose we wish to calculate the quantities

$1^2, 2^2, 3^2, \ldots, 10^2$

or a table of squares of integers from 1 to 10. This calculation can be described to the computer as calculating N^2, where the variable N assumes, one at a time, each of the values 1,2,3,. . .,10. Here is a sequence of Applesoft BASIC statements which accomplishes the calculations.


This is the first value of N
This is the last value of N
10 FOR N=1 TO 10
20 PRINT N^2
30 NEXT N
40 END
lines 10-20-30 repeated
10 times

The sequence of statements 10, 20, and 30 forms a loop. When the computer encounters the **FOR** statement, it sets N equal to 1 and continues executing the statements. Statement 20 calls for printing N^2. Since N is equal to 1, we have N^2=1^2=1 and the computer will print a 1. Next comes statement

30, which calls for the next N. This instructs the computer to return to the **FOR** statement in 10, increase N to 2, and to repeat instructions 20 and 30. This time N^2=2^2=4. Line 20 then prints a 4. Line 30 says to go back to line 10 and increase N to 3 and so forth. Lines 10, 20, and 30 are repeated 10 times. After the computer executes lines 10, 20, and 30 with N=10, it will leave the loop and execute line 40.

Type in the above program and give the **RUN** command. The output will look like this:

```
1
4
9
16
25
36
49
64
81
100
]
```

TEST YOUR UNDERSTANDING 1 (answers on page 102)

a. Devise a loop allowing N to assume the values 3 to 77.
b. Write a program which calculates N^2 for N=3 to 77.

Let's modify the above program to include not only N^2 on each line of output but the value of N as well. To make the table easier to read, let's also add two column headings. The new program reads

```
10 PRINT "N","N^2"
20 FOR N=1 TO 10
30 PRINT N,N^2
40 NEXT N
50 END
```

The output now looks like this:

```
N       N^2
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
10      100
]
```

TEST YOUR UNDERSTANDING 2 (answer on page 102)

What would happen if we change the number of line 10 to 25?

Let us now illustrate some of the many uses of loops by means of some examples.

Example 1. Write an Applesoft BASIC program to calculate $1+2+3+\ldots+100$.

Solution. Let us use a variable S (for sum) to contain the sum. Let us start S at 0 and use a loop to successively add to S the numbers 1,2,3,. . .,100. Here is the program.

```
10 S=0
20 FOR N=1 TO 100
30 S=S + N
40 NEXT N
50 PRINT S
60 END
```

These instructions repeated 100 times

When we enter the loop the first time, S=0 and N=1. Line 30 then replaces S by S + N, or 0 + 1. Line 40 sends us back to line 20 where the value of N is now set equal to 2. In line 30, S (which is now 0 + 1) is replaced by S + N, or 0 + 1 + 2. Line 40 now sends us back to line 20 where N is now set equal to 3. Line 30 then sets S equal to 0 + 1 + 2 + 3. Finally, on the 100th time through the loop, S is replaced by 0 + 1 + 2 + . . . + 100, the desired sum.

The variable S is an *accumulator variable* because it serves to accumulate the total as we progress. At any point during the calculation it holds the current subtotal. This type of accumulator device is extremely useful in a wide variety of applications. If we run the program, we derive the output

```
5050
]
```

TEST YOUR UNDERSTANDING 3 (answer on page 102)

Write an Applesoft BASIC program to calculate

$101+102+\ldots+110$

TEST YOUR UNDERSTANDING 4 (answer on page 102)

Write an Applesoft BASIC program to calculate and display the numbers $2,2^2,2^3,\ldots,2^{20}$.

Example 2. Write a program to calculate the sum

$$1\times2 + 2\times3 + 3\times4 + \ldots + 49\times50$$

Solution. We let the sum be contained in the variable S as we did in Example 1. The quantities to be added are just the numbers N*(N+1) for N=1,2,3,. . .,49. Here is our program.

```
10 S=0
20 FOR N=1 TO 49
30 S=S + N*(N+1)
40 NEXT N
50 PRINT S
60 END
```

Freezing The Output On The Screen

Example 3. You borrow $7,000 to buy a car. You finance the balance for 36 months at an interest rate of one percent per month. Your monthly payments are $232.50. Write a program which computes the amount of interest each month, the amount of the loan repaid, and the balance owed.

Solution. Let B denote the balance owed. Initially, we have B equal to 7,000 dollars. At the end of each month let us compute the interest (I) owed for that month, namely .01*B. For example, at the end of the first month, the interest owed is .01*7000.00= $70.00. Let P=232.50 to denote the monthly payment, and let R denote the amount repaid out of the current payment, or R=P − I.

For example, at the end of the first month, the amount of the loan repaid is 232.50 −70.00=162.50. The balance owed may then be calculated as B − R. At the end of the first month, the balance owed is 7000.00 − 162.50=6837.50. This problem is similar to others which we have already discussed. Here is a program which performs these calculations.

```
10 PRINT "INTEREST","BALANCE"
20 B=7000
25 P=232.50
30 FOR M=1 TO 36:REM M IS MONTH NUMBER
40 I=.01*B :REM CALCULATE INTEREST FOR MONTH
50 R=P - I :REM CALCULATE REPAYMENT
60 B=B - R :REM CALCULATE NEW BALANCE
70 PRINT I,B
80 NEXT M
90 END
```

Attempt to run this program. You will notice that it runs, but it is pretty useless because the screen will not contain all of the output. Most of the output goes flying by before you can read it.

There are several methods for remedying this situation. One such method is to use the CONTROL-C sequence as described in Section 3.5 as the output scrolls by on the screen. This will pause execution of the program and

freeze the contents of the screen. To resume execution and unfreeze the screen, type **CONT** and hit the **RETURN** key. The output will begin to scroll again. Using this technique requires some manual dexterity, plus it is not possible to guarantee where the scrolling will stop.

Another way to freeze the output on the screen is by using the **CONTROL-S** sequence. When you simultaneously hit these two keys, the output will stop scrolling by. To resume, you can hit any key at all. This method may be a little easier to use, and you don't obtain the BREAK message which you receive when using the **CONTROL-C**. However, it has the same problem, namely that you cannot guarantee where the scrolling will stop.

TEST YOUR UNDERSTANDING 5

RUN the program of Example 3 and practice freezing the output on the screen. Try doing this using both the **CONTROL-C** sequence with CONT to resume and **CONTROL-S** hitting any key to resume. It may take several runs before you are comfortable with the procedure.

A better way to freeze the output on the screen is to use the **STOP** statement in your program. The **STOP** statement will cause execution to pause until the word **CONT** is typed in.

Let's use this technique to control the screen size by printing only 12 months of data at one time. This amount of data will fit since the screen contains 24 lines. We will use a second loop to keep track of 12 month periods. The variable for the new loop will be Y (for "years"), and Y will go from 0 to 2. The month variable will be M as before, but now M will go only from 1 to 12. The month number will now be 12*Y + M (the number of years plus the number of months). Here is the revised program.

```
10 B=7000
15 P=232.50
20 FOR Y=0 TO 2
30 PRINT "INTEREST","BALANCE"
40 FOR M=1 TO 12
50 I=.01*B
60 R=P - I: REM ONE 12 MONTH PERIOD
70 B=B - R
80 PRINT I,B
90 NEXT M
100 STOP :REM HALTS EXECUTION
110 HOME :REM CLEAR SCREEN
120 NEXT Y :REM GOES TO NEXT 12 MONTHS
130 END
```

In line 100 we use the **STOP** statement. This causes the computer to stop execution of the program. The computer remembers where it stops, and all values of the variables are preserved. The **STOP** statement also leaves the contents of the screen unchanged. You can take as long as you wish to examine the data on the screen.

When you are ready for the program to continue, type **CONT**. The computer will resume where it left off. The first instruction it encounters is in line 110. **HOME** clears the screen, and the computer goes on to the next value of Y—the next 12 months of data. Here is a copy of the output. The underlined statements are those you type.

```
RUN
INTEREST        BALANCE
70              6837.5
68.375          6673.375
66.73375        6507.0875
65.0760875      6340.18484
63.4018484      6171.08669
61.7108669      6000.29755
60.0029755      5827.80053
58.2780053      5653.57854
56.5357854      5477.61432
54.7761432      5299.89047
52.9989047      5120.38937
51.2038937      4939.09326

BREAK IN 100
]
CONT

INTEREST        BALANCE
49.3909326      4755.98419
47.5598419      4571.04404
45.7104404      4384.25448
43.8425448      4195.59702
41.9559702      4005.05299
40.0505299      3812.60352
38.1260352      3618.22955
36.1822955      3421.91185
34.2191185      3223.63097
32.2363097      3023.36728
30.2336728      2821.10095
28.2110095      2616.81196

BREAK IN 100
]
CONT

INTEREST        BALANCE
26.1681196      2410.48008
24.1048008      2202.08488
22.0208488      1991.60573
19.9160573      1779.02179
17.7902179      1564.31201
15.6431201      1347.45513
13.4745513      1128.42968
11.2842968      907.213974
```

```
9.07213974      683.7856113
6.83786113      458.123975
4.58126         230.205214
2.30205215      7.26658106E-03
BREAK IN 100
]
```

The data in the output is carried out to ten figures even though the problem deals with dollars and cents.

Note the balance listed at the end of month 36. It is in scientific notation. The −03 indicates that the decimal point is to be moved three places to the left. The number listed is .00726658106 or about .72 cents (less than one cent)! The computer shifted to scientific notation since the usual notation requires more than ten digits. The computer made the choice of which form of the number to display.

In Section 3.8 we learned how to round numbers to any desired number of decimal places using the **INT** function. This will not work with large numbers which must be displayed in exponential format. Because this is not a problem for us here, we can round the numbers. To round to two decimal places, add line 75 and change line 80 as follows:

```
75 I1=INT(I*100 + .5)/100: B1=INT(B*100 + .5)/100
80 PRINT I1,B1
```

In Chapter Six we will learn a more convenient way to accomplish this.

Example 4. Write a program which generates 10 random numbers lying in the interval from 5 to 8.

Solution. Let us build up the desired function in two steps. We start from the function RND(1), which generates numbers from 0 to 1. First we adjust for the length of the desired interval. From 5 to 8 is 3 units, so we multiply RND(1) by 3. The function 3*RND(1) generates numbers from 0 to 3. Now we adjust for the starting point of the desired interval, namely 5. By adding 5 to 3*RND(1), we obtain numbers lying between 0+5 and 3+5, that is, between 5 and 8. Thus, 3*RND(1) + 5 generates random numbers between 5 and 8. Here is the program required.

```
10 FOR J=1 TO 10
20 PRINT 3*RND(1) + 5
30 NEXT J
40 END
```

Using Loops To Create Delays

By using a loop we can create a delay during the execution of a program. Consider the following sequence of instructions.

```
10 FOR N=1 TO 3000
20 NEXT N
```

This loop doesn't do anything! However, the computer must repeat instructions 10 and 20 three thousand times! This may seem like a lot of work for you, but not for a computer. To get a feel for the speed at which the computer works, time this sequence of instructions. Such a loop may be used as a delay.

For example, when you wish to keep some data on the screen without stopping the program, just build in a delay. Here is a program which prints two screens of text. A delay is imposed to give a person time to read the first screen

```
10 PRINT "THIS IS A PROGRAM TO DISPLAY SALES"
20 PRINT "FOR THE YEAR TO DATE"
30 FOR N=1 TO 5000
40 NEXT N: REM DELAY LOOP
50 HOME
60 PRINT "YOU MUST SUPPLY THE PARAMETERS:"
70 PRINT "PRODUCT, TERRITORY, VOLUME"
80 END
```

Example 5. Use a loop to produce a blinking display for a security system.

Solution. Suppose that your security system is tied in with your computer, and the system detects that an intruder is in your warehouse. Let us print out the message

```
SECURITY SYSTEM DETECTS INTRUDER-ZONE 2
```

For attention, let us blink this message on and off by alternately printing the message and clearing the screen.

```
10 FOR N=1 TO 2000
20 PRINT:PRINT:PRINT
30 PRINT "SECURITY SYSTEM DETECTS INTRUDER-ZONE 2"
40 FOR K=1 TO 150
50 NEXT K
60 HOME
70 FOR K=1 TO 150
80 NEXT K
90 NEXT N
100 END
```

The loop in 30-40 is a delay loop to keep the message on the screen a moment. Line 60 turns the message off, but the **PRINT** statement in line 30 almost immediately turns it back on. The message will blink 2000 times. In section 5.5 we will see another way to cause your message to blink.

TEST YOUR UNDERSTANDING 6 (answer on page 102)

Write a program which blinks your name on the screen 500 times, leaving your name on the screen for a loop of length 50 each time.

In all of our loop examples, the loop variable increased by one with each repetition of the loop. It is also possible to have the loop variable change by any amount. For example, the instructions

```
10 FOR N=1 TO 5000 STEP 2
.
.
.
1000 NEXT N
```

define a loop in which N jumps by 2 for each repetition, so N will assume the values

```
1,3,5,7,9,...,4999
```

Similarly, use of STEP .5 in the above loop will cause N to advance by .5 and assume the values

```
1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, ... , 5000
```

It is even possible to have a negative step. In this case the loop variable will run backwards. For example, the instructions

```
10 FOR N=100 TO 1 STEP -1
.
.
.
100 NEXT N
```

will "count down" from N=100 to N=1 one unit at a time.

TEST YOUR UNDERSTANDING 7 (answers on page 102)

Write instructions to allow N to assume the following sequences of values

a. 95,96.7,98.4,. . .,112
b. 200,199.5,199,. . .,100

Exercises (answers on page 313)

Write Applesoft BASIC programs to compute the following quantities.

1. $1^2+2^2+3^2+\ldots+25^2$
2. $1 + (1/2) + (1/2)^2 + \ldots + (1/2)^{10}$
3. $1^3+2^3+3^3+\ldots+10^3$
4. $1 + (1/2) + (1/3) + \ldots + (1/100)$
5. Write a program to compute N^2, N^3 for N=1,. . .,12. The format of your output should be as follows:

```
N     N^2     N^3
1
2
3
.
.
.
12
```

6. Suppose you have a car loan whose current balance is $4,000.00. The monthly payment is $125.33 and the interest is one percent per month on the unpaid balance. Make a table of the interest payments and balances for the next 12 months.
7. Suppose you deposit $1,000 on January 1 of each year into a savings account paying 10 percent interest. The interest is computed on January 1 of each year, based on the balance for the preceding year. Calculate the balances in the account for each of the next 15 years.
8. A stock market analyst predicts that Tyro Computers, Inc. will achieve a 20 percent growth in sales in each of the next three years, but profits will grow at a 30 percent annual rate. Last year's sales were $35 million and last year's profits were $5.54 million. Project the sales and profits for the next three years, based on the analyst's prediction.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, 3, 4, 6, and 7

1: a.
```
10 FOR N=3 TO 77
 .
 .
100 NEXT N
```
b.
```
10 FOR N=3 TO 77
20 PRINT N^2
30 NEXT N
40 END
```

2: The heading
```
N      N^2
```
would be printed before each entry of the table.

3:
```
10 S=0
20 FOR N=101 TO 110
30 S=S+N
40 NEXT N
50 PRINT S
60 END
```

4:
```
10 FOR N=1 TO 20
20 PRINT 2^N
30 NEXT N
40 END
```

6:
```
10 FOR N=1 TO 500
20 PRINT "<YOUR NAME>"
30 FOR K=1 TO 50
40 NEXT K
50 HOME
60 NEXT N
70 END
```

7: a.
```
10 FOR N=95 TO 112 STEP 1.7
```
b.
```
20 FOR N=200 TO 100 STEP -.5
```

5.2 Application 5-1: Mortgage Payment Calculations

Recent years and high interest rates have seen a great deal of creativity in the way major purchases, particularly houses, have been financed. One historically typical way in which such purchases were financed was via a mortgage. The lender and borrower agree on the interest rate to be charged and the number of years for which the loan will be outstanding. The monthly payments are then calculated to pay off the loan with interest in that time frame.

If we adopt the following symbols:

A = amount borrowed
I = annual interest rate
N = number of payments per year (usually 12)
Y = number of years for loan

then we can let

MI = interest rate per payment equals annual rate over number of payments per year (I/N)
N1 = total number of payments to be made equals number of payments per year times number of years (N*Y)

and the required payment can be calculated as

$$\frac{A * MI}{1 - (1 + MI)^{-N1}}$$

This payment consists partly of a payment of interest and partly of some payment of the principal or the amount borrowed. In the early years of a mortgage, most of the payment is interest because the outstanding balance is still very high. As the amount outstanding decreases, less interest is required and so more of the payment goes toward reducing the balance. The part of the payment which goes toward interest is calculated as the period interest rate (usually monthly) times the outstanding balance.

Because of the volatility in interest rates, many lenders have become unwilling to give long-term, fixed-rate mortgages. One solution is to give a mortgage in which the payments are geared to the normal long-term pay-

ment schedule (15, 20, or 30 years), but instead of allowing the mortgage to run that long, the lender requires that the balance of the loan actually be paid in a much shorter time (two to five years). The final payment is usually quite large and is referred to as a balloon payment.

In this application, we will write a program to calculate the required payment and, for each payment, find the interest paid, contribution to principal, and new loan balance. The program should also allow for balloon payments.

The major blocks in the solution procedure are:

initialize:
get amount of mortgage (A),
number of years on which payment is based (Y),
number of payments per year (N),
annual interest rate (I)
total number of payments to be made (for balloon)

calculate:
calculate monthly interest (MI)
get number of payments without a balloon (N1)
find payment to be made each period
write out payment

write table:
set current balance (OB) to amount of loan
loop for each payment except the last
calculate interest as MI times OB

print out results
endloop

last payment:
payment equals old balance plus interest
contribution to principal is payment minus interest
current balance is old current balance minus contribution
print results

From this solution procedure we should be able to construct the BASIC program:

```
10 HOME:PRINT "THIS CALCULATES A MORTGAGE PAYMENT SCHEDULE"
20 INPUT "AMOUNT OF MORTGAGE? ";A
30 INPUT "NUMBER OF YEARS ON WHICH PAYMENT IS BASED? ";Y
40 INPUT "NUMBER OF PAYMENTS PER YEAR? ";N
50 INPUT "ANNUAL INTEREST RATE IN DECIMAL? ";I
60 INPUT "BALANCE TO BE PAID ON PAYMENT # ";B
70 IF B > N*Y THEN PRINT "MAXIMUM NUMBER OF PAYMENTS IS "
   ;N*Y: GOTO 60
100 N1=N*Y: MI=I/N
110 P=(A * MI)/ (1 - (1 + MI)^(-N1) )
120 P=INT(P*100 + .05)/100
130 HOME: PRINT "FOR YOUR MORTGAGE OF ";A
140 PRINT:PRINT "THE PAYMENT IS ";P
150 PRINT:PRINT "TO BE MADE ";N;" TIMES PER YEAR"
```

```
200 REM PRINT HEADINGS FOR TABLE
210 PRINT:PRINT:PRINT
220 PRINT "PYMT";TAB(14);"PRNCPL";TAB(23);"INTRST";TAB(30);
    "OUTSTNDG"
230 PRINT "NUM";TAB(6);"PYMNT";TAB(14);"CONT";TAB(23);"AMT";
    TAB(30);"BALANCE"
300 PRINT:OB=A
310 FOR M=1 TO B - 1
320   IA=OB * MI
330   PC=P - IA: OB=OB - PC
340   IA=INT(IA*100 + .05)/100:PC=INT(PC*100 + .05)/100:
      OB=INT(OB*100 + .05)/100
350   PRINT M;TAB(6);P;TAB(14);PC;TAB(23);IA;TAB(30);OB
360 NEXT M
380 M=B : P=OB + IA: P=INT(P*100 + .05)/100
390   PC=P - IA: OB=OB - PC
400   IA=INT(IA*100 + .05)/100:PC=INT(PC*100 + .05)/100:
      OB=INT(OB*100 + .05)/100
410 PRINT: PRINT "THE LAST PAYMENT IS PAYMENT NUMBER ";M
420 PRINT: PRINT "AMOUNT OF LAST PAYMENT IS ";P
430 PRINT "INTEREST AMOUNT=";IA
440 PRINT "ENDING BALANCE IS ";OB
500 END
```

If we run the program with:

amount of mortgage = $30000
number of years on which mortgage payment is based = 15
number of payments per year = 12
annual interest rate = .12
balance to be paid on payment number 36 (3 year balloon)

then the output is:

```
FOR YOUR MORTGAGE OF 30000

THE PAYMENT IS 360.05

TO BE MADE 12 TIMES PER YEAR

PYMT        PRNCPL INTRST OUTSTNDG
NUM  PYMNT  CONT   AMT    BALANCE

1    360.05 60.05  300    29939.95
2    360.05 60.65  299.4  29879.3
3    360.05 61.25  298.79 29818.04
4    360.05 61.87  298.18 29756.17
5    360.05 62.48  297.56 29693.68
6    360.05 63.11  296.93 29630.56
7    360.05 63.74  296.3  29566.81
8    360.05 64.38  295.66 29502.42
9    360.05 65.02  295.02 29437.39
10   360.05 65.67  294.37 29371.71
11   360.05 66.33  293.71 29305.37
12   360.05 66.99  293.05 29238.37
```

```
 13   360.05 67.66  292.38 29170.7
 14   360.05 68.34  291.7  29102.35
 15   360.05 69.02  291.02 29033.32
 16   360.05 69.71  290.33 28963.6
 17   360.05 70.41  289.63 28893.18
 18   360.05 71.11  288.93 28822.06
 19   360.05 71.82  288.22 28750.23
 20   360.05 72.54  287.5  28677.68
 21   360.05 73.27  286.77 28604.4
 22   360.05 74     286.04 28530.39
 23   360.05 74.74  285.3  28455.64
 24   360.05 75.49  284.55 28380.14
 25   360.05 76.24  283.8  28303.89
 26   360.05 77.01  283.03 28226.87
 27   360.05 77.78  282.26 28149.08
 28   360.05 78.55  281.49 28070.52
 29   360.05 79.34  280.7  27991.17
 30   360.05 80.13  279.91 27911.03
 31   360.05 80.94  279.11 27830.09
 32   360.05 81.74  278.3  27748.34
 33   360.05 82.56  277.48 27665.77
 34   360.05 83.39  276.65 27582.37
 35   360.05 84.22  275.82 27498.14

THE LAST PAYMENT IS PAYMENT NUMBER 36

AMOUNT OF LAST PAYMENT IS 27773.96
INTEREST AMOUNT=275.82
ENDING BALANCE IS 0
```

5.3 Working With Tabular Data

We have discussed the notion of a variable and used variable names like

AA, B1, CZ, W0

Unfortunately, the supply of variables available to us is not sufficient for many programs. As we shall see in this chapter, there are relatively innocent programs which require hundreds or even thousands of variables.

To meet the needs of such programs, Applesoft BASIC allows for the use of **subscripted variables**. Such variables are used constantly in mathematical work and are identifed by numbered subscripts attached to a letter. For instance, here is a list of 1000 variables as they might appear in a mathematical work:

$A_1, A_2, A_3, \ldots, A_{1000}$

The numbers used to distinguish the variables are called **subscripts**.

The Applesoft BASIC language allows definition of variables to be distinguished by subscripts, but since the computer has difficulty placing the numbers in the traditional position, they are placed in parentheses on the same

line as the letter. For example, the above list of 1000 different variables would be written in Applesoft BASIC as

A(1),A(2),A(3),. . .,A(1000)

Remember, variable A(1) is not the same as the variable A1. You may use both of them in the same program and Applesoft BASIC will interpret them as being different.

A subscripted variable, also called an **array**, is really a group of variables with a common letter identification, distinguished by different integer "subscripts." For instance, the above group of variables would constitute the subscripted variable A( ). A helpful analogy is to think of an array as a 'family' of variables each with the same 'family name' just as you view families of people: the Jones family is a group of related persons each with the surname Jones.

Just as each member of the Jones family is a distinct individual, the members of an array are distinct variables. We distinguish one member of the Jones family from the next because they have different first names. Similarly, we distinguish different array members because they have different subscripts.

It is often useful to view a subscripted variable as a table. The subscripted variable A( ) considered above can be viewed as providing the following table of information:

A(1)
A(2)
A(3)
.
.
.
A(1000)

As shown here, the subscripted variable defines a table consisting of 1000 rows. Row number J contains a single entry, namely, the value of the variable A(J). The first row contains the value of A(1), the second the value of A(2), and so forth.

The array shown is a table consisting of 1000 rows and a single column. The Apple II Computer allows you to consider more general arrays.

For example, consider the following financial table which records the monthly income for January, February, and March from each of a chain of four computer stores:

	Store #1	Store #2	Store #3	Store #4
Jan.	1258.38	2437.46	4831.90	987.12
Feb.	1107.83	2045.68	3671.86	1129.47
March	1298.00	2136.88	4016.73	1206.34

This table has three rows and four columns. Its entries may be stored in the computer as a set of 12 variables:

A(1,1) A(1,2) A(1,3) A(1,4)

A(2,1) A(2,2) A(2,3) A(2,4)
A(3,1) A(3,2) A(3,3) A(3,4)

This array of variables is very similar to a subscripted variable except there are now two subscripts. The first subscript indicates the row number, and the second subscript indicates the column number; the variable A(3,2) is in the third row, second column.

A collection of variables such as that given above is called a **two-dimensional array** or a **doubly-subscripted variable**. Each setting of the variables in such an array defines a tabular array. For example, if we assign the values

A(1,1) = 1258.38, A(1,2) = 2437.46,
A(1,3) = 4831.90,

and so forth, we will have the table of earnings from the computer store chain.

So far we have only considered numeric arrays—arrays whose variables can assume only numerical values. It is also possible to have arrays with variables that assume string values. (Recall that a string is a sequence of characters: letter, numeral, punctuation mark, or other printable keyboard symbol.) Here is an array which can contain string data:

A$(1)
A$(2)
A$(3)
A$(4)

The dollar signs indicate that each of the variables of the array is a string variable. If we assign the values

A$(1) = "SLOW", A$(2) = "FAST", A$(3) = "FAST", A$(4) = "STOP"

the array is just the table of words

SLOW
FAST
FAST
STOP

Similarly, the employee record table

Social Security Number	Age	Sex
178654775	38	M
345861023	29	F
789257958	34	F
375486595	42	M
457696064	21	F

may be stored in an array of the form B$(I,J) where I assumes any one of the values 1, 2, 3, 4, or 5 (I is the row) and J assumes any one of the values 1,2,3 (J = the column). In this example B$(1,1) has the value "178654775", B$(1,2) has the value "38", B$(1,3) has the value "M", and so forth.

Because each member of the array is known by the array name and the name of the array defines the type of variable to be stored, each member of

the array must be the same variable type. For example, B$(I,J) refers to an array of string variables; A1(I) is an array of real numeric variables; Z%(J) is an array of integer variables.

The Apple II Computer even allows you to have arrays which have three, four, or even more subscripts. Consider again the computer store chain array introduced above. Suppose we had one such array for each of ten consecutive years. This collection of data could be stored in a three-dimensional array of the form C(I,J,K) where I and J represent the row and column, just as before, and K represents the year (K could assume the values 1,2,3,. . .,10.).

An array may involve any number of dimensions up to 88. The subscripts corresponding to each dimension may assume values from 0 to 32767. For all practical applications, any size array is permissible.

You must inform the computer of the sizes of the arrays you plan to use in a program. This allows the computer to allocate memory space to house all the values. To specify the size of an array, use a **dimension (DIM) statement.**

For example, to define the size of the subscripted variable A(J), J=1,. . .,1000, we insert the statement

```
10 DIM A(1000)
```

in the program. This statement informs the computer to expect variables A(0),A(1),. . .,A(1000) in the program and that it should set aside memory space for 1001 variables. In the absence of further instructions from you, Applesoft BASIC begins all subscripts at 0. If you wish to use A(0), fine. If not, ignore it.

You need not use all the variables defined by a **DIM** statement. In the case of the **DIM** statement above, you might actually use only the variables A(1),. . .,A(900). It is possible to use up all of the memory space, that is the RAM, which you have available, but this is not usually a problem unless you are using very large sets of data. For most of the things you are likely to want to do, it is best to make sure that you have defined enough variables; otherwise, you could be in trouble.

In the case of the subscripted variable above, your program might make use of the variable A(1001). This will create an error condition. Suppose that this variable is used first in line 570. When you attempt to run the program, the computer will report

```
? BAD SUBSCRIPT ERROR IN 570
```

and execution of the program will be halted. To fix the error merely redo the **DIM** statement to accommodate the undefined subscript.

To define the size of a two-dimensional array, use a **DIM** statement of the form

```
10 DIM A(5,4)
```

This statement defines an array A(I,J) where I can assume the values 0,1,2,3,4,5 and J can assume the values 0,1,2,3,4. Arrays with three or more subscripts are defined similarly.

TEST YOUR UNDERSTANDING 1 (answers on page 120)

Here is an array:

```
12  645.80
148 489.75
589 12.89
487 14.50
```

a. Define an appropriate subscripted variable to store this data.
b. Define an appropriate **DIM** statement.

It is possible to dimension several arrays with one **DIM** statement. The dimension statement

```
10 DIM A(1000), B$(5), A(5,4)
```

defines the array A(0),. . .,A(1000), the string array B$(0),. . .,B$(5) and the two-dimensional array A(I,J) , I= 0,. . .,5; J=0,. . .,4.

TEST YOUR UNDERSTANDING 2 (answer on page 120)

Suppose your program uses a 9 x 2 array A$(I,J), a 9 x 4 array B$(I,J), and a 9 x 5 array C(I,J). Write an appropriate DIM statement(s).

Giving Values To Array Members

We know how to set aside memory space for the variables of an array. We must now take up the problem of assigning values to these variables. We could use individual assignment statements, but with 1000 variables in an array this could lead to an unmanageable number of statements. There are more convenient methods which make use of loops. The following examples illustrate some of these methods.

Example 1. Define an array A(J) , J=1,2,. . .,1000 and assign the following values to the variables of the array:

```
A(1)=2, A(2)=4, A(3)=6, A(4)=8, ...
```

Solution. We wish to assign each variable a value equal to twice its subscript. That is, we wish to assign A(J) the value 2*J. To do this we use a loop:

```
10 DIM A(1000)
20 FOR J=1 TO 1000
30 A(J)=2*J
40 NEXT J
50 END
```

This example assigned values via an assignment statement. Note that the program ignores the variable A(0). Like any numeric variable which has not been assigned a value, it has the value 0. String variables, and that includes members of a string array, are automatically initialized to a blank.

TEST YOUR UNDERSTANDING 3 (answer on page 120)

Write a program which assigns the variables A(0), . . ., A(30) the values A(0)=0, A(1)=1, A(2)=4, A(3)=9, . . . (Note we are assigning the square of the subscript value.).

Example 2. Define an array corresponding to the employee record table given on page 108. Input the values given and print the table on the screen.

Solution. Our program will print the headings of the columns and then ask for the table entries one row at a time. We will store the entries in the array B$(I,J) where I is one of 1,2,3,4,5 and J is one of 1,2,3. We dimension the array as B$(5,3).

```
5 DIM B$(5,3)
10 FOR I=1 TO 5
20 INPUT "SS#,AGE,SEX"; B$ (I,1),B$(I,2),B$(I,3)
30 NEXT I: PRINT "SOC.SEC. #","AGE","SEX"
40 FOR I=1 TO 5
50 PRINT B$(I,1),B$(I,2),B$(I,3)
60 NEXT I:END
```

In this example values were assigned via an **INPUT** statement.

The following example illustrates the use of **DATA** statements in assigning values to an array.

Example 3. Suppose that the monthly electricity costs of a certain family are as follows:

Jan.	$89.74	Feb.	$95.84	March	$79.42
Apr.	78.93	May	72.11	June	115.94
July	158.92	Aug.	164.38	Sep.	105.98
Oct.	90.44	Nov.	89.15	Dec.	93.97

Write a program calculating the average monthly cost of electricity and writing out the month numbers of those months with higher than average cost.

Solution. The general approach can be seen in the following:

```
structure
  loop for each of 12 months (M)
    initialize: get cost for month M
    accumulate: add costs in T to obtain a total for year
  endloop
    average: calculate average cost (AV) by dividing by 12
             and print it
  loop for each of 12 months (M)
```

```
    compare: if cost in month M > AV then print month number
  endloop
```

We see here that we have two loops with some processing in between. Because we must use the values of monthly cost in each of the loops, we need to store them in an array when we initialize. Let us unceremoniously dump all of the numbers shown above into **DATA** statements at the end of the program. Arbitrarily, let's start the **DATA** statements at line 1000, with **END** at 2000. This gives us plenty of room. Let us create the array C(M), M = 1,2,. . .,12 and set C(M) equal to the cost of electricity in the Mth month.

```
10 DIM C(12)
15 REM LINES 20-40 ASSIGN VALUES TO C(M)
20 FOR M=1 TO 12
30   READ C(M)
60   T=T + C(M): REM T ACCUMULATES THE SUM OF THE C(M)
70 NEXT M
80 AV=T/12 : REM DIVIDE SUM BY 12
90 PRINT "THE AVERAGE MONTHLY COST OF ELECTRICITY IS ";AV
100 PRINT "THE MONTHS WITH HIGHER THAN AVERAGE COSTS ARE"
110 PRINT: PRINT "MONTH";SPC(5);"COST"
120 FOR M=1 TO 12
130  IF C(M) > AV THEN PRINT SPC(2);M;SPC(7);C(M)
140 NEXT M
1000 DATA 89.74, 95.84, 79.42, 78.93, 72.11, 115.94
1010 DATA 158.92, 164.38, 105.98, 90.44, 89.15, 93.97
2000 END
```

The following program could be helpful in preparing the payroll of a small business. It combines all three methods of giving values to array members.

Example 4. A small business has five employees. Here are their names and hourly wages

Name	Hourly Wage
Joe Polanski	7.75
Susan Greer	8.50
Allan Cole	8.50
Betsy Palm	6.00
Herman Axler	6.00

Write a program which accepts as input hours worked for the current week and then calculates the current gross pay and the amount of Social Security tax to be withheld from their pay. (Assume Social Security tax amounts to 6.70 percent of gross pay.)

Solution. In this problem, we need to initialize the names of the employees, their hourly wage rates, and their hours worked. In outlining the solution procedure we will keep in mind that name and wage-rate data are relatively stable. Therefore, we would probably want to keep these in DATA statements. Hours worked may change from pay period to pay period, so this information will be INPUT. Our solution procedure recognizes this distinction by keeping these as two separate blocks.

```
loop for all five employees (J)
  read data: get employee name and wage rate
  input data: get hours worked
  gross wages: hours worked times wage rate
  social security tax: .067 times gross wages
endloop
print headings
loop for each employee (J)
  output: print name, gross wage, social security tax
endloop
```

Let us keep the hourly wage rates and names in two arrays, called A(J) and B$(J) respectively, where J=1,2,3,4,5. We can't use a single two-dimensional array for this data since the names are string data and the hourly wage rates are numerical. (Recall that Applesoft BASIC does not allow us to mix the two kinds of data in an array.) The number of hours worked during the current week will be stored in the array C(J); J=1,2,3,4,5. Gross wages are stored in the array D(J); J=1,2,3,4,5. The amount of Social Security tax to be withheld is in the array E(J); J=1,2,3,4,5. Here is the program.

```
10 DIM A(5),B$(5),C(5),D(5),E(5)
20 FOR J=1 TO 5
30   READ B$(J),A(J)
40   PRINT "TYPE CURRENT HOURS OF ", B$(J)
50   INPUT C(J)
60   D(J)=A(J)*C(J)
70   E(J)=.0670*D(J)
80 NEXT J
90 PRINT "EMPLOYEE","GROSS WAGES";SPC(5);"SS.TX"
100 FOR J=1 TO 5
110  PRINT B$(J),D(J);TAB(33);E(J)
120 NEXT J
200 DATA JOE POLANSKI, 7.75, SUSAN GREER, 8.50
210 DATA ALLAN COLE, 8.50, BETSY PALM, 6.00
220 DATA HERMAN AXLER, 6.00
1000 END
```

If you plan to dimension an array, always insert the DIM statement before the variable first appears in your program. Otherwise, the first time Applesoft BASIC comes across the array, it will assume that the subscripts go from 0 to 10. If it subsequently comes across a **DIM** statement, it will think you are changing the size of the array in the midst of the program which is not allowed. If you try to change the size of an array in the middle of a program, you will get an error:

```
? REDIM'D ARRAY ERROR
```

It is possible to leave the size of the array as a variable in your program as long as that variable is assigned a value prior to the **DIM** statement. For example:

```
10 INPUT "HOW MANY ITEMS ARE THERE? ";N
20 DIM A(N)
```

Here the size of the array is determined when the program is executed. The value of N input is used to dimension the array.

When the computer is first turned on or is reset, all variables (including those in arrays) are cleared. All numeric variables are set equal to 0. All string variables are set equal to the null string (the string with no characters in it). If you wish to return all variables to this state during the execution of a program, use the command **CLEAR**.

For example, when the computer encounters the command

```
570 CLEAR
```

it will reset all the variables. The **CLEAR** command can be convenient if you wish to use the same array to store two different sets of information at two different stages of the program. After the first use of the array, you could prepare for the second use by executing a **CLEAR**.

A Roulette Game Example

In this example, we will combine the use of arrays and loops with the random number generation which we discussed in Section 3.8 in order to simulate the action of a roulette wheel.

A roulette wheel has 38 equally-spaced positions. These are numbered 1 through 36, 0 and 00. The wheel is balanced so that when it is spun, the ball has a chance to land on any of the 38 positions.

Example 5. Write a program which turns the computer into a roulette wheel. Let the computer keep track of bets and winnings for any number of players. For simplicity, assume that the only bets are on single numbers.

Solution. In our program we will represent the 38 positions as the numbers 1-38, with 37 corresponding to 0 and 38 corresponding to 00. A spin of the wheel will consist of choosing a random integer between 1 and 38.

We begin by asking for the number of players. Each player has an account stored in the array A(J); players begin by purchasing chips to place in their accounts. For a typical spin of the wheel, the program will ask for bets by each player. A bet will consist of a number (1-38) and an amount bet. These will be stored in the arrays B(J) and C(J), respectively. The wheel will then spin. The program will determine the winners and losers. A payoff for a win is 32 times the amount bet. At the end of each spin, the program then asks if another play is desired.

The solution procedure is:

```
     initialize: get the number of players (N)
loop for each player
     input chips: get number of chips for each player
endloop
loop for each game
     loop for each player
     bets: get number to be played and amount bet
```

```
      endloop
      spin wheel: generate random integer between 1 and 38
           (X)
      loop for each player
           determine wins and losses
                if number bet=X then player wins 32 times
                   amount bet, add to player's account
                 if not, then subtract amount bet from
                 account
      endloop
      loop for each player
           output: player number and account amount
      endloop
end game loop
```

Here is the program.

```
10 INPUT "NUMBER OF PLAYERS";N
20 DIM A(N),B(N),C(N)
25 REM LINES 30-60 ALLOW PLAYERS TO PURCHASE CHIPS
30 FOR J=1 TO N : REM FOR EACH OF THE PLAYERS
40   PRINT "PLAYER "; J
50   INPUT "HOW MANY CHIPS"; A(J)
60 NEXT J
100 PRINT "LADIES AND GENTLEMEN!"
105 PRINT "PLACE YOUR BETS PLEASE!"
110 FOR J=1 TO N : REM FOR EACH OF THE PLAYERS
120  PRINT "PLAYER "; J
130  INPUT "NUMBER, AMOUNT"; B(J),C(J):REM INPUT BET
140 NEXT J
200 X=INT(38*RND(1) + 1): REM SPIN THE WHEEL
210 REM LINES 210-300 DISPLAY THE WINNING NUMBER
220 PRINT "THE WINNER IS NUMBER "; X
300 REM: LINES 310-590 DETERMINE WINNINGS AND LOSSES
310 FOR J=1 TO N : REM FOR EACH PLAYER
320  IF X=B(J) THEN 400
325  GOTO 330
330  A(J)=A(J) - C(J): REM PLAYER J LOSES. DEDUCT BET
340  PRINT "PLAYER ";J ;" LOSES"
350  GO TO 420
400  A(J)=A(J) + 32*C(J): REM PLAYER J WINS. ADD WINNINGS
410  PRINT "PLAYER ";J;" WINS "; 32*C(J); " DOLLARS"
420 NEXT J
430 PRINT "PLAYER BANKROLLS"
440 PRINT
450 PRINT "PLAYER", "CHIPS"
460 FOR J=1 TO N
470  PRINT J,A(J)
480 NEXT J
500 INPUT "DO YOU WISH TO PLAY ANOTHER ROLL(Y/N)";R$
510 HOME
520 IF R$="Y" THEN 100
530 PRINT "THE CASINO IS CLOSED. SORRY!"
600 END
```

You should try a few spins of the wheel. The program is fun as well as instructive. Although the program allows you to bet more chips than you have, we will leave it to the exercises to add in a test for enough chips to cover the bet. You could also build lines of credit into the game!

Another Look At Name The State Capitals

In Section 3.6 we developed a program in which the computer would repeatedly give the name of a state and then would ask us to name the capital of that state. The data was read from DATA statements. The difficulty here is that sometimes the player comes to depend upon the order in which the states are presented, memorizing a sequence of capital names instead of associating a given state with the correct capital.

Now that we have learned about arrays, we can add a little twist to that program. We can begin by loading all of the state names and capitals into an array. Then we can use the random number generator to obtain random values between 1 and 52. This will indicate which state should be quizzed.

Here is the program as developed earlier.

```
5 HOME
10 READ S$: IF S$="NONE" THEN 150
20 READ C$
30 PRINT "WHAT IS THE CAPITAL OF ";S$
40 G=1: S=S+ 1
50 PRINT: PRINT: INPUT G$
60 IF G$=C$ THEN PRINT: PRINT: PRINT "CORRECT, ";:
     PT=PT - 4*G: TS=TS+PT: GOTO 100

80 IF G < 3 THEN PRINT "SORRY, TRY AGAIN ": G=G+1:
     GOTO 50
90 PRINT "SORRY, ";:
100 PRINT "THE CAPITAL OF ";S$;" IS ";C$
110  PRINT: PRINT: PRINT "YOUR TOTAL SCORE IS ";TS;
     "AFTER ";S;" STATES"
120 INPUT "DO YOU WANT TO SEE MORE STATES? (Y/N)? ";A$
130 IF A$="Y" THEN HOME: GOTO 10
140 IF A$ <> "N" THEN 120
150  PRINT "THERE ARE NO MORE STATES": PRINT: PRINT:
          PRINT: "YOUR FINAL SCORE IS ";TS: PRINT:
160 PRINT "HIGHEST POSSIBLE SCORE IS ";S*10: PRINT
170 PRINT: PRINT "IF YOU WISH TO PLAY AGAIN, TYPE RUN"
180 END
1000 DATA ILLINOIS,SPRINGFIELD
1010 DATA WASHINGTON, OLYMPIA
1020 DATA PENNSYLVANIA, HARRISBURG
          .
          .
          .
9000 DATA NONE
```

To introduce our element of randomness we will use the following procedure:

```
initialize: load state and capital names into an array (N$)
loop until user asks to quit
     random: choose a random integer between 1 and 52
     test: proceed with quiz as previously determined
endloop
output: report final score
```

We will define a two-dimensional array N$(52,2) and place the name of state I in N$(I,1) and its capital in N$(I ,2).

```
5 HOME: DIM N$(52,2)
10 FOR I=1 TO 52: READ N$(I,1),N$(I,2): NEXT I
20 I=INT(52*RND(1) + 1): S$=N$(I,1): C$=N$(I,2)
30 PRINT "WHAT IS THE CAPITAL OF ";S$
40 G=1: S=S+ 1
50 PRINT: PRINT: INPUT G$
60 IF G$=C$ THEN PRINT: PRINT: PRINT "CORRECT, ";:
     PT=PT - 4*G: TS=TS+PT: GOTO 100
80 IF G < 3 THEN PRINT "SORRY, TRY AGAIN ": G=G +1:
     GOTO 50
90 PRINT "SORRY, ";
100 PRINT "THE CAPITAL OF ";S$;" IS ";C$
110  PRINT: PRINT: PRINT "YOUR TOTAL SCORE IS ";TS;
     "AFTER ";S;" STATES"
120 INPUT "DO YOU WANT TO SEE MORE STATES? (Y/N)? ";A$
130 IF A$="Y" THEN HOME: GOTO 20
140 IF A$ <> "N" THEN 120
150 PRINT: PRINT "YOUR FINAL SCORE IS ";TS: PRINT:
160 PRINT "HIGHEST POSSIBLE SCORE IS ";S*10: PRINT
170 PRINT: PRINT "IF YOU WISH TO PLAY AGAIN, TYPE RUN"
180 END
1000 DATA ILLINOIS,SPRINGFIELD
1010 DATA WASHINGTON, OLYMPIA
1020 DATA PENNSYLVANIA, HARRISBURG
          .
          .
          .
```

Exercises (answers on page 314)

For each of the following tables, define an appropriate array and determine the appropriate DIM statement.

1.
```
5
2
1.7
4.9
11
```

2.
```
1.1  2.0  3.5
1.7  2.4  6.2
```

3.
```
JOHN
MARY
SIDNEY
```

4. 1 2 3

5.
```
RENT            575.00
UTILITIES       249.78
CLOTHES         174.98
CAR             348.70
```

6. Each of the following programs assigns values to the variables of an array. Determine which values are assigned.

a.
```
10 DIM A(10)
20 FOR J=1 TO 10
30 READ A(J)
40 NEXT J
50 DATA 2,4,6,8,10,12,14,16,18,20
100 END
```

b.
```
10 DIM A(3),B(3)
20 FOR J=0 TO 3
30 READ A(J), B(J)
40 NEXT J
50 DATA 1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9
60 END
```

c.
```
10 DIM A(3), B(3)
20 READ A(0),B(0)
30 READ A(1),B(1)
40 RESTORE
50 READ A(2),B(2)
60 READ A(3),B(3)
70 DATA 1,2,3,4,5,6,7,8
80 END
```

d.
```
10 DIM A(3,4)
20 FOR I=1 TO 3
30 FOR J=1 TO 4
40 READ A(I,J)
50 NEXT J
60 NEXT I
70 DATA 1,2,3,4,5,6,7,8,9,10,11,12
80 END
```

7. Display the following array on the screen.

	Receipts	
	Store #1	Store #2
1/1-1/10	57,385.48	89,485.45
1/11-1/20	39,485.98	76,485.49
1/21-1/31	45,467.21	71,494.25

8. Write a program that displays the array of Exercise 7 along with totals of the receipts from each store.

9. Expand the program in Exercise 8 so it calculates and displays the totals for each day. (Your screen will not be wide enough to display the sales data and daily totals, so display the daily totals in a separate display.)
10. Devise a program which keeps track of the inventory of an appliance store chain. Store the current inventory in an array of the form

	Store #1	Store #2	Store #3	Store #4
Refrig.				
Stove				
Air Cond.				
Vacuum				
Disposal				

Your program should: (1) input the inventory corresponding to the beginning of the day; (2) continually ask for the next transaction—the store number and the number of appliances of each item sold; and (3) in response to each transaction, update the inventory array and redisplay it on the screen.

11. Modify the roulette program of Example 5 to check that a player has enough chips to cover the bet.
12. Modify the roulette program of Example 5 to allow for a $100 line of credit for each player.
13. Here is a table of Federal Income Tax Withholding of weekly wages for an individual claiming one exemption. Assume that each of the employees of Example 4 claims a single exemption. Modify the given program so it correctly computes Federal Withholding and the net amount of wages (that is, the total after Federal Withholding and Social Security are deducted).

Wages at Least	But Less Than	Tax Withheld
200	210	29.10
210	220	31.20
220	230	33.80
230	240	36.40
240	250	39.00
250	260	41.60
260	270	44.20
270	280	46.80
280	290	49.40
290	300	52.10
300	310	55.10
310	320	58.10
320	330	61.10
330	340	64.10
340	350	67.10

14. Here is a set of 24 hourly temperature reports as compiled by the National Weather Service. Write a program to compute the average temperature for the last 24 hours. Let your program respond to a

query concerning the temperature at a particular hour. (For example, what was the temperature at 2:00 PM?)

	AM	PM
12:00	10	38
1:00	10	39
2:00	9	40
3:00	9	40
4:00	8	42
5:00	11	38
6:00	15	33
7:00	18	27
8:00	20	22
9:00	25	18
10:00	31	15
11:00	35	12

15. Make up a list of ten names. Write a program which will pick four of the names at random.(This is a way of impartially assigning a nasty task!)

ANSWERS TO TEST YOUR UNDERSTANDING 1, 2, and 3

1: a. `A(I,J), I=1,2,3,4; J=1,2` b. `DIM A(4,2)`

2:
```
DIM A$(9,2),B$(9,4),C(9,5)
```

3:
```
10 DIM A(30)
20 FOR J=0 TO 30
30 A(J)=J^2
40 NEXT J
50 END
```

5.4 Application 5-2: The Return On An Investment

One of the difficult problem areas many people face in both their personal lives and in business is the evaluation of alternative investments. A way of looking at this problem is to set up some minimum rate of return level. Frequently this is called a **hurdle rate** or **discount rate**. Investment opportunities which meet this rate are considered to be worthy of further evaluation.

In some cases the return is fixed and stated at the time of investment. For example, if you buy a certificate of deposit or place money in a savings account, you are given the interest rate which will be paid. This is your rate of return on that investment. In other cases neither the cash flows over the years to come nor the rate of return are known. How do you measure the return on this kind of investment?

The usual means for estimating the return (we can only estimate because so much is unknown) is to estimate the yearly cash flows over the life of the investment. A yearly cash flow is just the amount of cash which will flow into or out of our possession because of the investment during the year.

For example, let's say we were considering the purchase of some property we intended to rent out and eventually resell after five years. The initial cost of the property is $120,000, and we estimate we should be able to net the following in rental collections less expenses over the next five years.

year	cash flow
1	1200
2	1700
3	2000
4	2400
5	2400

In year five we will sell the property at a price we estimate to be $140,000.

We decide that we are not interested in the investment unless we can obtain at least a 12 percent rate of return. The interest rate or rate of return you would like to obtain from the investment is a subjective judgment which you would probably base on the rate you might obtain from other alternative investments with some adjustment for risk. For example, if you can obtain 7.5 percent on a savings account which is a certain return, you might want 12 or 15 percent from a more risky alternative.

Once the cashflows are estimated and the minimum desired rate of return for the investment is selected, we need to know if this investment meets the criterion. Is this investment worthwhile?

One method used to answer this is called the **Present Value** method. This method takes each yearly cashflow and calculates what we would have to invest today to obtain that cashflow. It then totals these amounts to indicate to us what would have to be put into an account at the desired rate in order to be able to withdraw the same amounts each year. If this amount is less than the amount we are thinking of investing in the project, the return on the project does not meet the desired rate. For our rental property example, if the Present Value is less than the $120,000 we are thinking of investing, then the investment is too costly. We will not make a 12 percent return.

The technique used to calculate the Present Value is the calculation of a **discount factor** for each year. The factor for year J is equal to 1 plus the interest rate raised to the Jth power. The estimated cashflow for each year is divided by the discount factor for that year to obtain the present value of that cashflow. When all discounted cashflows are added together, we have the present value of the investment. This can then be compared to the amount of investment itself.

In our example, we have:

investment = 120000, discount rate = .12

year	cashflow	discount factor	present value
1	1200	1.12	1071.43

2	1700	1.25	1355.23
3	2000	1.40	1423.56
4	2400	1.57	1525.24
5	142400	1.76	80801.58
present value of investment			86177.04

In this case the present value is less than the 120,000 investment, so we know the rate of return is less than 12 percent.

This means if you could put $86,177.04 on deposit for 12 percent interest per year, you would be able to withdraw $1200 after one year, $1700 after two years, $2000 in year three, $2400 in year four, and $142400 the last year. Essentially, you can obtain the same cashflows for less of an investment.

Let's structure a solution procedure for the computer.

```
initialize: get initial investment C(0)
            get life of investment in years (N)
            get yearly cashflows C(Y), Y=1,. . .,N
            get discount rate (I)

calculate present value (PV)

output: write out initial investment, cashflows, present
            value of investment
        if present value >=investment then rate of return
            exceeds discount rate
        if present value < investment then rate of return
            is less than discount rate
```

The **calculate present value** block needs to be specified further. We take each cashflow and discount it to the present. We add each of the discounted cashflows to an accumulator which serves as a subtotal. Here is the expanded logic for this section:

```
set accumulator to add discounted cashflows (PV)
loop for each year (Y)
     calculate discount factor as (1+I)^Y
     divide cashflow by discount factor
     add discounted cashflow to accumulator
endloop
```

Adding these two parts together, the complete solution procedure is:

```
initialize: get initial investment C(0)
            get life of investment in years (N)
            get yearly cashflows C(Y), Y=1,. . .,N
            get discount rate (I)
calculate present value (PV)
     set accumulator to add discounted cashflows (PV)
     loop for each year (Y)
          calculate discount factor as (1+I)^Y
          divide cashflow by discount factor
          add discounted cashflow to accumulator
     endloop
```

```
output: write out initial investment, cashflows, present
          value of investment
        if present value >=investment then rate of return
          exceeds discount rate
        if present value < investment then rate of return
          is less than discount rate
```

With this specification, we can write a program.

```
10 PRINT "PROGRAM TO CALCULATE PRESENT VALUE"
20 INPUT "WHAT IS INITIAL INVESTMENT? ";C(0)
30 INPUT "WHAT IS LIFE OF INVESTMENT? ";N
40 DIM C(N)
50 FOR Y=1 TO N
60    PRINT "CASHFLOW FOR YEAR ";Y;" IS ";:INPUT C(Y)
70 NEXT Y
80 INPUT "WHAT IS DESIRED RATE OF RETURN (DECIMAL)? ";I

1000 PV=0: REM PRESENT VALUE CALCULATION
1010 FOR Y=1 TO N
1020   F=(1+I)^Y
1030   DC=C(Y)/F
1040   PV=PV + DC
1050 NEXT Y

1100 PRINT "INITIAL INVESTMENT IS ";C(0)
1110 PRINT "DISCOUNT RATE IS ";I
1120 PRINT: PRINT "YEAR","CASHFLOW"
1130 FOR Y=1 TO N
1140   PRINT SPC(2);Y,C(Y)
1150 NEXT Y
1160 PR=INT (PV*100 + .05)/100 : PRINT: PRINT
1170 PRINT "THE PRESENT VALUE OF THIS INVESTMENT IS ";PR
1180 PRINT: PRINT
1190 IF PV>=C(0) THEN PRINT "DESIRED RATE IS MET"
1200 IF PV<C(0) THEN PRINT "RETURN IS LESS THAN DESIRED"

9000 END
```

The program follows the solution procedure outlined, but it also contains a lot of embellishment to allow for ease of use and clarity of output, including rounding to two decimal places in line 1160.

5.5 Adding Pizazz To Your Programs

In this section we will introduce a few techniques you can use to make the output of your programs a little flashier and the input process a little easier.

We have already introduced the **HOME** command which automatically clears the screen and sets the cursor at the upper left corner. We can also place the cursor in other places on the screen using the **VTAB** and **HTAB** statements.

The **VTAB** statement determines the cursor placement vertically on the screen, that is, in a particular row. The rows on the screen are numbered 1 to 24 with 1 the top row. Try the following:

```
HOME
VTAB(15)
VTAB(3)
VTAB(24)
```

When you type the first of these commands you should see the screen clear and the cursor move to the upper left of the screen. (By the way, if you have an Apple IIe, you can clear the screen in immediate mode by hitting the ESC key and then the P.) The **VTAB(15)** moves the cursor to the middle of the screen; the **VTAB(3)** moves it back up to the third row; finally, the **VTAB(24)** moves it down to the lower left corner. The **VTAB** statement can be used with or without the parentheses around the row number.

Horizontal movement is controlled by the **HTAB** command. The screen is numbered 1 to 40 from the left to the right. The **HTAB** statement will only work if you are inputting or printing something after placing the cursor. Try this:

```
HOME
HTAB(15): PRINT "HI"
HTAB(3): PRINT "WELL"
HTAB(40): PRINT "THERE"
```

This sequence of commands clears the screen and sets the cursor at the upper left corner. Each successive statement prints the desired word at the indicated column. These statements can be used within the program to place the cursor anywhere on the screen that you wish.

The normal way of having things displayed on the screen is in white on black (or white on green or something similar depending on your video display unit). You can ask the computer to display things in inverse, or black on white, by using the command **INVERSE**. Try the following program:

```
10 HOME: INVERSE
20 PRINT "THIS IS AN EXAMPLE OF INVERSE PRINTING"
30 PRINT "NOW LET'S GO BACK TO NORMAL"
40 NORMAL
50 PRINT "THIS IS MORE LIKE IT"
```

The **NORMAL** command turns the inverse video off.

Similarly, you can ask the computer to blink on and off by using the **FLASH** command. Try this:

```
10 HOME: FLASH
20 PRINT "THIS IS AN EXAMPLE OF FLASHING"
30 PRINT "NOW LET'S GO BACK TO NORMAL"
40 NORMAL
50 PRINT "THIS IS MORE LIKE IT"
```

You can use any combination of these special commands that you wish. For example,

```
10 HOME: FLASH: INVERSE
20 PRINT "THIS IS AN EXAMPLE OF FLASHING INVERSE"
30 NORMAL: INVERSE: PRINT "NOW LET'S JUST HAVE INVERSE"
40 NORMAL: FLASH: PRINT "NOW FLASH ONLY"
50 NORMAL: PRINT "NOW ALL SYSTEMS NORMAL"
```

Another handy input device in addition to the **INPUT** and **READ-DATA** we already discussed is the **GET** command. In this command, although we ask for input from the keyboard, it is not necessary for the user to hit the RETURN key. The disadvantage is that the **GET** command can only input one value (a character or single digit) at a time. It also does not echo the input on the screen: the user does not see what has been accepted by the computer.

In the following example the computer asks for a simple Y or N (yes or no) answer.

```
10 PRINT "DO YOU UNDERSTAND THIS (Y/N)?": GET A$
```

Run this program. You will see that it is only necessary to type in a letter (preferably a Y or N as these are the correct options) without hitting RETURN. The computer does not display what you typed in. You can remedy this with another PRINT statement:

```
10 PRINT "DO YOU UNDERSTAND THIS (Y/N)? ";:GET A$: PRINT A$
```

Now the answer is displayed on the screen. Because of the semicolon you inserted after the first PRINT statement, the answer is printed on the same line as the question. This is important because the user can see whether he or she typed in the correct response.

The **GET** command can be very useful in certain kinds of programs, but it is usually a good idea to be consistent in the way that you request input. If you use the **GET**, use it consistently so you develop a habit of inputting the data in one way. This eliminates many input errors.

Another technique which is useful in helping people use our program correctly is **menu selection**. This can be used whenever the program provides for several options.

For example, assume we have a program which allows us to select among four different routines. We might wish to display a screen which looks like this:

```
CHOOSE ONE OF THE FOLLOWING:
     1-FIRST ROUTINE
     2-SECOND ROUTINE
     3-THIRD ROUTINE
     4-FOURTH ROUTINE
TYPE IN ONE NUMBER
```

This is called a **menu**. It can be implemented in BASIC quite simply.

```
10 HOME
20 PRINT "CHOOSE ONE OF THE FOLLOWING:"
30 PRINT
40 PRINT TAB(5);"1--FIRST ROUTINE "
```

```
50 PRINT TAB(5);"2--SECOND ROUTINE "
60 PRINT TAB(5);"3--THIRD ROUTINE "
70 PRINT TAB(5);"4--FOURTH ROUTINE "
80 PRINT: PRINT "TYPE IN ONE NUMBER ";: GET A: PRINT A
```

Note that the ability to use a GET statement here is limited to a menu which does not require two digit numbers.

If you are using menus for input, you will generally branch to a particular point in your program depending upon the routine selected. This can be accomplished with IF . . . THEN statements, but there is an easier way.

The ON . . . GOTO statement provides for multiple branches. Its form is

```
ON <expression> GOTO n1, n2, n3, n4
```

The expression is evaluated. If it equals 1 then program control branches to statement number n1; if its value is 2, program control branches to statement number n2, and so forth. In our example, if the first routine was located at statement 1000, the second at 3000, the third at 450, and the fourth at 3400, the statement would be

```
ON A GOTO 1000, 3000, 450, 3400
```

Let's augment the above program to see how this works.

```
10 HOME
20 PRINT "CHOOSE ONE OF THE FOLLOWING:"
30 PRINT
40 PRINT TAB(5);"1--FIRST ROUTINE "
50 PRINT TAB(5);"2--SECOND ROUTINE "
60 PRINT TAB(5);"3--THIRD ROUTINE "
70 PRINT TAB(5);"4--FOURTH ROUTINE "
80 PRINT: PRINT "TYPE IN ONE NUMBER ";: GET A: PRINT A
90 ON A GOTO 1000, 3000, 450, 3400
450 PRINT "ROUTINE 3":END
1000 PRINT "ROUTINE 1":END
3000 PRINT "ROUTINE 2":END
3400 PRINT "ROUTINE 4":END
```

Run this program and try inputting different values. If you try to input something out of range (below 1 or above 4), the control merely drops to the next statement following the **ON . . . GOTO**. We must provide some security against someone entering an invalid number (in Section 9.1 we will discuss how to protect against entering a non-numeric character).

We can use the commands we just learned to do this. Examine the following program. See if you can tell what it is doing before you actually run it.

```
10 HOME
20 PRINT "CHOOSE ONE OF THE FOLLOWING:"
30 PRINT
40 PRINT TAB(5);"1--FIRST ROUTINE "
50 PRINT TAB(5);"2--SECOND ROUTINE "
60 PRINT TAB(5);"3--THIRD ROUTINE "
70 PRINT TAB(5);"4--FOURTH ROUTINE "
80 PRINT: PRINT "TYPE IN ONE NUMBER ";: GET A: PRINT A
```

```
90 ON A GOTO 1000, 3000, 450, 3400
100 FLASH: VTAB(15): PRINT TAB(5);"ONLY VALUES FROM 1 TO 4
    ALLOWED": NORMAL
110 FOR K=1 TO 2000: NEXT K
120 VTAB(15): PRINT SPC(40)
130 VTAB(8): HTAB(20): PRINT " ": VTAB(8): HTAB(20): GET A:
    PRINT A: GOTO 90
450 PRINT "ROUTINE 3":END
1000 PRINT "ROUTINE 1":END
3000 PRINT "ROUTINE 2":END
3400 PRINT "ROUTINE 4":END
```

One final thing you can do to make your screen output a little fancier is to change the speed at which it is printed out. This is done via the **SPEED** command. The speed can be changed from 0 to 255 by a statement such as

```
SPEED=50
```

Try the following program with different values for speed.

```
10 HOME
20 INPUT "SPEED (0-255)? ";S
30 SPEED=S
40 PRINT "THIS IS ONLY A TEST"
50 END
```

Exercises (answers on page 316)

1. Write a BASIC program which will request that the user input his or her social security number. Have the program write out the correct number of spaces and insert the dashes as follows

   ```
   SOCIAL SECURITY NUMBER? ___-__-____
   ```

   Then move the cursor to the first position of input and GET each number one at a time. Remember to position the cursor where you want it. Also remember that you must PRINT the character after the GET statement because the GET does not echo on the screen. You will have to use semicolons at the end of the print statements to keep the numbers on the same line.
2. Modify the program of Exercise 1 so that the underlines flash until the user types in the values.

5.6 Application 5-3: Investment Analysis System

In Section 5.4 we examined the present value method for evaluating an investment. Present value is only one of several possibilities. One very straightforward method is calculating the **payback period**, or the number of years before the cash coming in from the investment covers the cost of the investment itself ignoring interest. If we spend $500 to purchase some device designed to save

money on utility bills, and we expect to save $100 per year, the payback period is 5 years because in five years we will have saved enough to cover the $500 which we spent. Obviously, this does not take into account any earnings we might have realized from investing that $500 elsewhere. (In fact, we can calculate payback period assuming some rate of interest, but this destroys the simplicity of the payback method which is its main virtue.)

Another method, called the **internal rate of return**, is closely related to the present value method. With the present value method the user specifies some desired rate of return or hurdle rate. The present value is then calculated using that rate. With the internal rate of return method, the investment and cashflows are given, and the rate of return which makes the present value of the cashflows equal to the investment is sought. The internal rate of return is a measure of the return on that investment project.

We would like to develop a program which would give the user the option of calculating the payback period, present value, or internal rate of return on an investment. We will use the menu technique discussed in the previous section to allow selection of these options. The major blocks of the program are:

```
initialize:  get initial investment C(0)
             get life of investment in years (N)
             get yearly cashflows C(Y), Y=1,...,N

echo:  write out the data which was input

menu:  select payback period, present value, internal rate
       of return, or exit system

payback period:  calculate payback period
                 write out answer
                 return to menu

present value:  get discount rate (I)
                calculate present value (PV)
                write out result
                return to menu

internal rate of return:  calculate internal rate of
                          return
                          write out answer
                          return to menu
```

The major blocks call for getting the input values and then displaying them on the screen, allowing the user to select an option, performing that option, and then returning to the menu for another selection. Several of these blocks require further detail. Let's work on them one by one.

Payback Period

To be sure we understand the problem, let's use a numerical example. Assume the investment is $2500 and the returns for each of the next seven years are

year	cash flow	cumulative receipts
1	350	350
2	500	850
3	600	1450
4	750	2200
5	750	2950
6	800	3750
7	650	4400

How soon have we recouped our investment? As you can see from the cumulative receipts column, we cover our outlay somewhere in year five. By the end of year four we have taken in only $2200, but by the end of year five our receipts are $2950, which is greater than the $2500 we invested. By the end of year four we have still not recouped $2500 - 2200 = $300. Given that we will collect $750 in year five, it will require approximately 300/750 = .4 of that fifth year to complete the recovery of investment. The total payback period then is 4.4 years.

How can we use the computer to find the payback period? We can continue adding in yearly cashflows until the accumulated amount is greater than the investment. At this point we know that we have gone too far. Thus, we can back up one year, see how much of the investment is not covered at the end of that year, and then calculate the proportion of the next year required to cover that portion.

Let's write a solution procedure for this block.

```
calculate payback period:
    set accumulator to add cashflows (S)
    loop until for each year (Y) or exit
    add next cashflow to accumulator
    if accumulator > investment then exit loop
    endloop
    if Y > N then investment is not recovered, print
         appropriate message, return to menu
    if Y <=N then set year to Y-1, reduce accumulator by
         cash flow in year Y,
         calculate remainder as investment - accumulator
         calculate proportion of year needed as remainder
              divided by cash flow in year Y + 1
              payback period is Y + proportion calculated
```

This is now in sufficient detail to allow us to write a program.

Present Value

We discussed present value in Section 5.4. We will need to make only minor changes to that solution procedure because the input data is part of the input to the whole system. The only input required specifically by the present value block is the desired rate of return or discount rate. Here is the revised procedure.

```
calculate present value:
    set accumulator to add discounted cashflows (PV)
    loop for each year (Y)
         calculate discount factor as (1+I)^Y
         divide cashflow by discount factor
         add discounted cashflow to accumulator
      endloop

write out answer:
    if present value >=investment then rate of return
         exceeds discount rate
    if present value < investment then rate of return is
         less than discount rate
```

Internal Rate of Return

When we used the present value approach, we assumed that there was some desired rate of return or hurdle rate. In the internal rate of return approach, we assume that we know only the initial investment and estimates of the cash flows over the life of the investment. The aim is to find the rate of return which makes the present value of those future cashflows equal to the initial investment. This is not an easy problem to solve without a computer. There is no general analytic approach which can be used, and therefore, the problem must be solved by trial and error. While this can be quite tedious by hand, it is a simple task for the computer.

The trial and error approach requires choosing a value for the interest rate, calculating the present value using that rate, and then, by noting the difference between the investment and the present value, choosing a new trial value for the rate. When the two amounts are equal, you have found the rate of return for that investment or the internal rate of return.

To search for the correct rate via a trial and error procedure, we need to know whether the new trial rate is larger or smaller than the one we just tried. To see how to make adjustments in the new trial value, it is necessary to look at a typical term in the summation for the present value. Such a term is

$$(\text{cashflow for year Y}) / (1 + \text{rate})^{Y}$$

This term is the quotient of two numbers or one number divided by another. If we increase the value of the denominator (the number on the bottom), we will decrease the value of the entire term. Consider the example $3/2 = 1.5$. If we increase the denominator to 4, we have $3/4 = .75$, a smaller number. Thus, if we wish to reduce the present value, we can do so by increasing the denominator, or discount factor, in each term.

The discount factor can be increased by increasing the trial rate of return. When we compare the present value and the investment, if the present value exceeds the investment, we must increase the trial rate; if the present value is less than the investment, we must decrease the rate.

Implementing a trial and error procedure on the computer is quite easy because the computer does not mind the tedious repetition. Even so, it would

not be practical to expect to find the rate of return which made the present value and the investment exactly equal. This would, at best, require many decimal places for the rate. In a practical sense it is not important what that rate is out to the fourth or fifth place, particularly since our cash flows are only estimates. We will use a stopping rule which says that when we have obtained a present value which is within 1% of the investment, we will consider the estimated rate of return to be close enough. The 1% figure is arbitrary and you may wish to use some other stopping rule.

Let's try to write our solution procedure in more detail now:

```
calculate internal rate of return:

initialize: calculate bounds for stopping rule as lower
          bound (L)=.99 * investment, upper bound (U)=
          1.01 * investment

loop until present value is between L and U
     choose rate: get trial value to use

     calculate present value

     compare: if present value > U then increase rate
              if present value < L then decrease rate
endloop
```

As you can see, one of the blocks in this procedure is to calculate the present value. This is exactly the same block as in the present value calculation, so we can use the same part of the program to implement this routine. We do need, however, to expand the ***choose rate*** block a little more. There are many ways that you can choose trial rates to use in your calculation. We will use a method called a **binary search technique**. We begin by setting upper and lower bounds on the value of the rate and then select a point half way between these two bounds.

For example, assume we arbitrarily say our program will handle rates of return between 0 and 100 percent, or .0 and 1.0 in decimal. These are the initial bounds on the interest rate. Our first trial value would then be at (0.0 + 1.0) /2 or 0.5. If the first trial indicated that 0.5 was too high, we would know the true rate was someplace between 0.0 and 0.5 and the next trial value would be (0.0 + 0.5)/2=0.25. Subsequent values could be found in a similar fashion: adjusting the upper or lower bounds on the interest rate and then choosing a value half way between them.

To write the solution procedure for this block, we need to make some adjustments to the ***initialize*** block and to the ***compare*** block. The complete procedure for **calculate internal rate of return** is:

```
calculate internal rate of return:

initialize: calculate bounds for stopping rule as lower
          bound (L)=.99 * investment, upper bound (U)=
          1.01 * investment
          set lower interest bound (LI) at 0, upper
          interest bound (UI) at 1.0

loop until present value is between L and U
```

```
      choose rate: set trial rate at (LI + UI)/2=I
      calculate present value

      compare: if present value > U then trial rate is too
               small, set new value of LI at I
               if present value < L then trial rate is too
               large, set new value of UI at I
endloop
```

The procedure for internal rate of return is now complete.

The Solution Procedure

Now we can put down the entire solution procedure and begin writing the BASIC program. The entire solution process is:

```
initialize:  get initial investment C(0)
             get life of investment in years (N)
             get yearly cashflows C(Y), Y=1,...,N

echo: write out the data which was input

menu:  select payback period, present value, internal rate
       of return, or exit system

payback period:

  calculate payback period:
    set accumulator to add cashflows (S)
    loop for each year (Y) or exit
      add next cashflow to accumulator
      if accumulator > investment then exit loop
    endloop
  if Y > N then investment is not recovered, print
    appropriate message, return to menu
  if Y <=N then set year to Y-1, reduce accumulator by
    cashflow in year Y, calculate remainder as investment -
    accumulator, calculate proportion of year needed as
    remainder divided by cashflow in year Y + 1 payback
    period is Y + proportion calculated
  write out answer
  return to menu

present value:
  get discount rate (I)
  calculate present value:
    set accumulator to add discounted cashflows (PV)
    loop for each year (Y)
      calculate discount factor as (1+I)^Y
      divide cashflow by discount factor
      add discounted cashflow to accumulator
    endloop

  write out answer:
```

```
  if present value >=investment then rate of return
    exceeds discount rate
  if present value < investment then rate of return is
    less than discount rate

  return to menu

internal rate of return:

  calculate internal rate of return:
    initialize: calculate bounds for stopping rule as lower
      bound (L)=.99 * investment, upper bound (U)=1.01 *
      investment set lower interest bound (LI) at 0, upper
      interest bound (UI) at 1.0

    loop until present value is between L and U
      choose rate: trial rate is (LI + UI)/2=I

      calculate present value
      compare: if present value > U then trial rate is too
        small, set new value of LI at I
               if present value < L then trial rate is too
        large, set new value of UI at I

    endloop
    write out answer
    return to menu
```

The BASIC Program

At this point we can write the BASIC program. To begin let's assign blocks of line numbers to correspond to each block in the solution procedure. The major blocks and line numbers assigned are

block	*line numbers*
initialize	1- 99
echo	100- 199
menu	500- 799
payback period	1000-2999
present value	3000-4999
internal rate of return	5000-7999

These assignments are made rather arbitrarily and should provide plenty of room for the development of each section of the program. By assigning line numbers ahead of time, we can work on each section independently without worrying about putting the pieces together later on.

Let's work on each section of the program one at a time. The **initialize** block is first and will look very much like the program written in Section 5.4.

BLOCK 1—initialize: get initial investment C(0)
get life of investment in years (N)
get yearly cashflows C(Y), Y=1,...,N

(line numbers 1-99)

```
1 REM INITIALIZE
10 HOME: VTAB(10):PRINT "INVESTMENT ANALYSIS SYSTEM":PRINT
20 PRINT:INPUT "WHAT IS INITIAL INVESTMENT? ";C
30 PRINT:INPUT "WHAT IS LIFE OF INVESTMENT (YEARS)? ";N
40 DIM C(N):C(0)=C: PRINT
50 FOR Y=1 TO N
60    PRINT "CASHFLOW FOR YEAR ";Y;" IS ";:INPUT C(Y)
70 NEXT Y
```

BLOCK 2—echo: write out the data which was input
(line numbers 100-199)

```
100 REM ECHO
110 HOME: PRINT "INITIAL INVESTMENT IS ";C(0)
130 PRINT: PRINT "YEAR","CASHFLOW"
140 FOR Y=1 TO N
150 PRINT SPC(2);Y,C(Y)
160 NEXT Y
170 VTAB(20): PRINT "HIT RETURN FOR MENU ":GET A$
```

BLOCK 3—menu: select payback period, present value, internal rate of return, display input, or exit system

(lines 500-799)

```
500 REM MENU SELECTION
510 HOME:VTAB(10)
520 PRINT "CHOOSE ONE OF THE FOLLOWING:"
530 PRINT
540 PRINT TAB(3);"1--DISPLAY INPUT DATA"
550 PRINT TAB(3);"2--CALCULATE PAYBACK PERIOD"
560 PRINT TAB(3);"3--CALCULATE PRESENT VALUE"
570 PRINT TAB(3);"4--CALCULATE INTRNL RATE OF RETURN"
580 PRINT TAB(3);"5--EXIT SYSTEM"
590 PRINT: PRINT "TYPE IN ONE NUMBER ";: GET A: PRINT A
600 ON A GOTO 100, 1000, 3000, 5000, 9000
610 FLASH: VTAB(15): PRINT TAB(5);"ONLY VALUES FROM 1 TO 5
     ALLOWED": NORMAL
620 FOR K=1 TO 2000: NEXT K
630 VTAB(15): PRINT SPC(40)
640 VTAB(9): HTAB(20): PRINT " ": VTAB(9): HTAB(20): GET A:
     PRINT A: GOTO 600
```

BLOCK 4—payback period:

calculate payback period:
  set accumulator to add cashflows (S)
  loop for each year (Y) or exit
    add next cashflow to accumulator
    if accumulator > investment then exit loop
  endloop
  if Y > N then investment is not recovered, print

appropriate message, return to menu
if Y <= N then set year to Y-1, reduce
accumulator by cashflow in year Y, calculate
remainder as investment - accumulator,
calculate proportion of year needed as
remainder divided by cashflow in year Y + 1
payback period is Y + proportion calculated

write out answer
return to menu

(line numbers 1000-2999)

```
1000 REM PAYBACK PERIOD CALCULATION
1010 S=0
1020 FOR Y=1 TO N
1030 S=S + C(Y)
1040 IF S > C(0) THEN 1100
1050 NEXT Y
1060 PRINT:PRINT "INVESTMENT IS NOT RECOVERED DURING
      STATED LIFE"
1070 PRINT "RATE OF RETURN IS NEGATIVE"
1080 GOTO 500
1100 Y=Y - 1: S=S - C(Y + 1)
1110 R=C(0) - S: P=R/C(Y + 1): PP=Y + P
1120 HOME: VTAB (12)
1130 PP=INT (PP*100 + .05)/100
1140 VTAB(12):PRINT "THE PAYBACK PERIOD FOR THIS INVESTMENT
      IS ";PP;"YEARS"
1150 VTAB(20): PRINT "HIT RETURN FOR MENU ":GET A$
1160 GOTO 500
```

BLOCK 5 — present value:
get discount rate (I)

calculate present value:
set accumulator to add discounted cashflows (PV)
loop for each year (Y)
calculate discount factor as (1+I)^Y
divide cashflow by discount factor
add discounted cashflow to accumulator
endloop

(line numbers 3000-4999)

```
3000 REM PRESENT VALUE CALCULATION
3010 HOME: VTAB(8):INPUT "WHAT IS DESIRED RATE OF RETURN
      (DECIMAL)? "; I
3020 PV=0
3030 FOR Y=1 TO N
3040 F=(1+I)^Y
3050 DC=C(Y)/F
3060 PV=PV + DC
3070 NEXT Y
```

Because we will also want to use this routine for the internal rate of return calculation, we will add another line to this routine which will send us back to the internal rate of return calculation if that was the option selected.

```
3100 IF A=4 THEN GOTO 6000
```

write out answer:
  if present value > = investment then rate of return
    exceeds discount rate
  if present value < investment then rate of return
    is less than discount rate
  return to menu

```
3200 HOME: PV=INT (PV*100 + .05)/100 : VTAB(12)
3210 PRINT "THE PRESENT VALUE OF THIS INVESTMENT IS ";PV
3220 PRINT: PRINT
3230 IF PV >=C(0) THEN PRINT "DESIRED RATE IS MET"
3240 IF PV < C(0) THEN PRINT "RETURN IS LESS THAN DESIRED"
3250 VTAB(20): PRINT "HIT RETURN FOR MENU ":GET A$
3260 GOTO 500
```

BLOCK 6—internal rate of return:
  calculate internal rate of return:
    initialize: calculate bounds for stopping rule as
      lower bound (L) = .99 * investment,
      upper bound (U) = 1.01 * investment
      set lower interest bound (LI) at 0,
      upper interest bound (UI) at 1.0

    loop until present value is between L and U
      choose rate: trial rate is (LI + UI)/2 = I
      calculate present value
      compare: if present value > U then trial rate
        is too small, set new value of LI at I
          if present value < L then trial rate
        is too large, set new value of UI at I
    endloop

  write out answer
  return to menu

```
5000 REM INTERNAL RATE OF RETURN CALCULATION
5010 HOME:FLASH:HTAB(10):PRINT "CALCULATING":NORMAL
5020 L=.99 * C(0): U=1.01 * C(0)
5030 LI=0: UI=1
5100 I=(LI + UI)/2
5110 GOTO 3020
6000 IF PV > U THEN LI=I: GOTO 5100
6010 IF PV < L THEN UI=I: GOTO 5100
6020 HOME: VTAB (12)
6030 IR=INT(I*100 + .05)/100
6040 PV=INT(PV*100 + .05)/100
6050 PRINT "THE INTERNAL RATE OF RETURN IS ";IR
```

```
6060 PRINT: PRINT "THE PRESENT VALUE AT THIS RATE IS ";PV
6070 VTAB(20): PRINT "HIT RETURN FOR MENU ":GET A$
6070 GOTO 500
```

Because we already had the program statements to calculate present value, all we had to do was to branch to statement 3020 to do the calculation and then return to the internal rate of return calculaion to see if we needed to repeat the calculation with a new trial value.

The only other statement we need in the program is the **END** statement

```
9000 END
```

The completed program is

```
1 REM INITIALIZE
10 HOME: VTAB(10):PRINT "INVESTMENT ANALYSIS SYSTEM":PRINT
20 PRINT:INPUT "WHAT IS INITIAL INVESTMENT? ";C
30 PRINT:INPUT "WHAT IS LIFE OF INVESTMENT (YEARS)? ";N
40 DIM C(N):C(0)=C: PRINT
50 FOR Y=1 TO N
60 PRINT "CASHFLOW FOR YEAR ";Y;" IS ";:INPUT C(Y)
70 NEXT Y
100 REM ECHO
110 HOME: PRINT "INITIAL INVESTMENT IS ";C(0)
120 PRINT: PRINT "YEAR","CASHFLOW"
130 FOR Y=1 TO N
140 PRINT SPC(2);Y,C(Y)
150 NEXT Y
160 VTAB(20): PRINT "HIT RETURN FOR MENU ":GET A$
500 REM MENU SELECTION
510 HOME:VTAB(10)
520 PRINT "CHOOSE ONE OF THE FOLLOWING:"
530 PRINT
540 PRINT TAB(3);"1--DISPLAY INPUT DATA"
550 PRINT TAB(3);"2--CALCULATE PAYBACK PERIOD"
560 PRINT TAB(3);"3--CALCULATE PRESENT VALUE"
570 PRINT TAB(3);"4--CALCULATE INTRNL RATE OF RETURN"
580 PRINT TAB(3);"5--EXIT SYSTEM"
590 PRINT: PRINT "TYPE IN ONE NUMBER ";: GET A: PRINT A
600 ON A GOTO 100, 1000, 3000, 5000, 9000
610 FLASH: VTAB(15): PRINT TAB(5);"ONLY VALUES FROM 1 TO 5
     ALLOWED": NORMAL
620 FOR K=1 TO 2000: NEXT K
630 VTAB(15): PRINT SPC(40)
640 VTAB(9): HTAB(20): PRINT " ": VTAB(9): HTAB(20): GET A:
     PRINT A: GOTO 600
1000 REM PAYBACK PERIOD CALCULATION
1010 S=0
1020 FOR Y=1 TO N
1030    S=S + C(Y)
1040    IF S > C(0) THEN 1100
1050 NEXT Y
1060 PRINT:PRINT "INVESTMENT IS NOT RECOVERED DURING
      STATED LIFE"
```

```
1070 PRINT "RATE OF RETURN IS NEGATIVE"
1080 GOTO 500
1100 Y=Y - 1: S=S - C(Y + 1)
1110 R=C(0) - S: P=R/C(Y + 1): PP=Y + P
1120 HOME: VTAB (12)
1130 PP=INT (PP*100 + .05)/100
1140 VTAB(12):PRINT "THE PAYBACK PERIOD FOR THIS INVESTMENT
     IS ";PP;"YEARS"
1150 VTAB(20): PRINT "HIT RETURN FOR MENU ":GET A$
1160 GOTO 500
3000 REM PRESENT VALUE CALCULATION
3010 HOME: VTAB(8):INPUT "WHAT IS DESIRED RATE OF RETURN
     (DECIMAL)? "; I
3020 PV=0
3030 FOR Y=1 TO N
3040 F=(1+I)^Y
3050 DC=C(Y)/F
3060 PV=PV + DC
3070 NEXT Y
3100 IF A=4 THEN GOTO 6000
3200 HOME: PV=INT (PV*100 + .05)/100 : VTAB(12)
3210 PRINT "THE PRESENT VALUE OF THIS INVESTMENT IS ";PV
3220 PRINT: PRINT
3230 IF PV>=C(0) THEN PRINT "DESIRED RATE IS MET"
3240 IF PV < C(0) THEN PRINT "RETURN IS LESS THAN DESIRED"
3250 VTAB(20): PRINT "HIT RETURN FOR MENU ":GET A$
3260 GOTO 500
5000 REM INTERNAL RATE OF RETURN CALCULATION
5010 HOME:FLASH:HTAB(10):PRINT "CALCULATING":NORMAL
5020 L=.99 * C(0): U=1.01 * C(0)
5030 LI=0: UI=1
5100 I=(LI + UI)/2
5110 GOTO 3020
6000 IF PV > U THEN LI=I: GOTO 5100
6010 IF PV < L THEN UI=I: GOTO 5100
6020 HOME: VTAB (12)
6030 IR=INT(I*100 + .05)/100
6040 PV=INT(PV*100 + .05)/100
6050 PRINT "THE INTERNAL RATE OF RETURN IS ";IR
6060 PRINT: PRINT "THE PRESENT VALUE AT THIS RATE IS ";PV
6070 VTAB(20): PRINT "HIT RETURN FOR MENU ":GET A$
6070 GOTO 500
9000 END
```

Executing the Program

Let's try running this program now using the same data that we used in Section 5.4 to try the present value program. This data is:

initial investment = 120000
cash flows are:

year	cashflow
1	1200
2	1700
3	2000
4	2400
5	142400

Let's run the program exercising each of the options. Here is the output:

```
INVESTMENT ANALYSIS SYSTEM

WHAT IS THE INITIAL INVESTMENT? 120000
WHAT IS THE LIFE OF INVESTMENT (YEARS)? 5
CASHFLOW FOR YEAR 1 IS ?1200
CASHFLOW FOR YEAR 2 IS ?1700
CASHFLOW FOR YEAR 3 IS ?2000
CASHFLOW FOR YEAR 4 IS ?2400
CASHFLOW FOR YEAR 5 IS ?142400

INITIAL INVESTMENT IS 120000

year    cash flow
 1          1200
 2          1700
 3          2000
 4          2400
 5        142400

HIT RETURN FOR MENU


CHOOSE ONE OF THE FOLLOWING:
1 -- DISPLAY INPUT DATA
2 -- CALCULATE PAYBACK PERIOD
3 -- CALCULATE PRESENT VALUE
4 -- CALCULATE INTRNL RATE OF RETURN
5 -- EXIT SYSTEM

TYPE IN ONE NUMBER 2

THE PAYBACK PERIOD FOR THIS INVESTMENT IS 4.04 YEARS
HIT RETURN FOR MENU

CHOOSE ONE OF THE FOLLOWING:
1 -- DISPLAY INPUT DATA
2 -- CALCULATE PAYBACK PERIOD
3 -- CALCULATE PRESENT VALUE
4 -- CALCULATE INTRNL RATE OF RETURN
5 -- EXIT SYSTEM
TYPE IN ONE NUMBER 3

WHAT IS THE DESIRED RATE OF RETURN (DECIMAL)
? .15
THE PRESENT VALUE OF THIS INVESTMENT IS
75814.13
```

```
RETURN IS LESS THAN DESIRED
HIT RETURN FOR MENU

CHOOSE ONE OF THE FOLLOWING:
1 -- DISPLAY INPUT DATA
2 -- CALCULATE PAYBACK PERIOD
3 -- CALCULATE PRESENT VALUE
4 -- CALCULATE INTRNL RATE OF RETURN
5 -- EXIT SYSTEM
TYPE IN ONE NUMBER 4

CALCULATING
THE INTERNAL RATE OF RETURN IS .04
THE PRESENT VALUE AT THIS RATE
IS 119688.18
HIT RETURN FOR MENU

CHOOSE ONE OF THE FOLLOWING:
1 -- DISPLAY INPUT DATA
2 -- CALCULATE PAYBACK PERIOD
3 -- CALCULATE PRESENT VALUE
4 -- CALCULATE INTRNL RATE OF RETURN
5 -- EXIT SYSTEM
TYPE IN ONE NUMBER 5
```

When you run this program, each of the separate pieces of output will appear on a fresh screen. The word CALCULATING which appears during the internal rate of return routine will flash so that the user knows that the computer is processing data.

5.7 Debugging and Testing

You have probably already discovered that there are many ways to make errors when writing programs. An error is sometimes called a "bug" in computer jargon. The process of finding these errors or "bugs" in a program is called ***debugging***. This can often be a ticklish task. Manufacturers of commercial software must regularly repair the bugs users discover in their programs!

In this section we will try to provide some means of dealing with the errors which come up and discuss some of the features which your Apple II Computer has to help detect bugs. Remember, when we use the term Apple II Computer in this book, we are referring to the Apple II Plus or the Apple IIe.

It may be useful to note there are basically three kinds of errors which can be distinguished: *syntax errors, execution errors,* and *logic errors.*

Syntax errors occur when there is a program line (or lines) which the computer is unable to understand due to an error in the way it was written. This may be due to a typing error, or it may be due to incorrect construction of a BASIC statement. In this case program execution ends with a message like

```
SYNTAX ERROR IN 530
```

To analyze the error you type

```
LIST 530
```

resulting in the display

```
530 LET Y=(X+2(X^2-2)
```

We note that there is an open parenthesis (without a corresponding close parenthesis). This is enough to trigger an error. We modify line 530 to read

```
530 LET Y=X+2(X^2-2)
```

We **RUN** the program again and find that there is still a syntax error in line 530! This is the frustrating part since not all errors are easy to spot. If you look closely at the expression on the right, you will note we have omitted the * to indicate the product of 2 and (X^2-2). This is a common mistake, especially for those familiar with the use of algebra. (In algebra, the product is usually indicated without any operation sign.) We correct line 530 again. (You may either retype the line or use the line editor as discussed in Section 3.7.)

```
530 LET Y=X+2*(X^2-2)
```

Now there is no longer a syntax error in line 530!

Sometimes the statements in our program are correctly constructed, but the computer cannot execute them for some reason. These can be called execution errors. In these cases the computer will stop execution and will provide an error message which should help you to see what is wrong. We saw examples of this kind of problem when we told the computer to **READ** but did not provide enough data or when we tried to redimension an array in the program. Perhaps we incorrectly provided data so that the computer tried to divide by zero. Each of these cases would lead to a stop in execution and an error message.

The computer can often help in this instance since it can recognize many of the most common errors. The computer will print an error message indicating the error type and the line number in which it occurred. This will allow you to **LIST** the indicated line and attempt to find the cause of the error. The appendix to this chapter contains a list of the most common error messages. For a complete list of error messages, refer to the **BASIC Programming Reference Manual**, Apple Publication #A2L0006. It should be noted that the computer error messages will tell you the kind of error which was encountered in trying to execute the program. Sometimes, however, the true source of that execution error is actually someplace else in the program. We will discuss some methods of tracking these problems down a little later.

Execution errors are frequently the result of some error in the logic used to construct the solution. Because of the nature of these errors the computer stops execution and provides an error message. The most serious logic errors are those in which the computer is able to execute the entire program and provide an answer, but the answer is not correct! Sometimes it is obvious that there is a problem, but this is not always the case.

The best way to protect yourself against logic errors is to systematically structure the problem solution as discussed in Chapter Four, but even this is not a guarantee. One useful technique which can help detect non-obvious logic errors is the use of test data. By running the program with data to which you already know the answer, you can gain some assurance that it is running correctly. If possible, the test data should cover each branch the program might take.

For example, consider a situation in which we have data on employees in an organization. This data includes the employee's name, hourly rate, and years of service. We want a program to write out the names of all employees who have been with the firm at least 15 years.

Here is the program.

```
10 PRINT "NAME","YEARS"
20 READ N$
30 IF N$="NOMORE" THEN END
40 READ Y
50 IF Y>=15 THEN PRINT N$,Y
60 END
100 DATA PETER,21
110 DATA MARY,15
120 DATA JOE,9
150 DATA NOMORE
```

There are several branches in this program, each of which can be tested. The test data includes one employee with more than 15 years, one with exactly 15 years, and one with fewer than 15 years. The last data line is the flag data. We know that if the program is correct, both PETER and MARY should be written out. This is, indeed, what happens.

Note the two values to be read in are in separate **READ** statements. This is so that we can avoid having to put in dummy data for the years of service. This is particularly helpful when there are many data items to be read in for each employee.

The use of test data is particularly important when arithmetic calculations are involved. The data used should be data for which the answer is known (which usually means it is very simple data!). Preferably, it should be constructed so that each possible branch is tested. This may be next to impossible in very complicated programs, but some of the techniques for program structure which we will discuss in the next chapter will ease this problem a bit.

Sometimes we will know something is wrong in the program, but we simply don't know what it is. There are a couple of useful approaches to isolating such a problem. The first is simply to put in some extra print statements to display the intermediate results. This will frequently pinpoint the problem.

Consider the following program.

```
10 HOME
20 PRINT "THIS PROGRAM COMPUTES AN AVERAGE"
30 PRINT "TYPE IN NUMBERS ONE AT A TIME"
```

```
40 PRINT "TYPE 0 WHEN FINISHED"
50 I=I + 1
60 INPUT N
70 IF N=0 THEN 100
80 S=S + N
90 GOTO 60
100 A=S/I
110 PRINT "THE AVERAGE IS ";A
```

Using the concept of test data, we can see if the program is working correctly by finding the average of 1 and 2 which we know should be 1.5 [(1+2)/2]. Running this program yields the following output:

```
THIS PROGRAM COMPUTES AN AVERAGE
TYPE IN NUMBERS ONE AT A TIME
TYPE 0 WHEN FINISHED
?1
?2
?0
THE AVERAGE IS 3
```

Because we are using test data, we know the answer is not 3, but 1.5. To try to locate the problem, let's add an extra print statement to see what values we are actually using in our computation. At line 105, let us print out the values of S and I which are used to compute the average by adding the line

```
105 PRINT S,I
```

We do not need fancy PRINT statements because they will be used only for debugging and will be discarded once the program is running correctly. If we rerun the program with this statement in it, we obtain the following:

```
THIS PROGRAM COMPUTES AN AVERAGE
TYPE IN NUMBERS ONE AT A TIME
TYPE 0 WHEN FINISHED
?1
?2
?0
3               1
THE AVERAGE IS 3
```

Now we can see that the sum seems to be correct, but the number of values in the sum is only one. We do not seem to be correctly counting the number of values. By examining the program we see the GOTO statement only sends us back to the input and does not allow for incrementing the counter. We can correct this by letting line number 90 say

```
90 GOTO 50
```

Rerunning the program we obtain the following:

```
THIS PROGRAM COMPUTES AN AVERAGE
TYPE IN NUMBERS ONE AT A TIME
TYPE 0 WHEN FINISHED
?1
```

```
?2
?0
3                3
THE AVERAGE IS 1
```

The total count is still not correct. The value of I should be 2, but we obtain 3. You may see what the difficulty is by examining the program which now looks like this:

```
10 HOME
20 PRINT "THIS PROGRAM COMPUTES AN AVERAGE"
30 PRINT "TYPE IN NUMBERS ONE AT A TIME"
40 PRINT "TYPE 0 WHEN FINISHED"
50 I=I + 1
60 INPUT N
70 IF N=0 THEN 100
80 S=S + N
90 GOTO 50
100 A=S/I
105 PRINT S,I
110 PRINT "THE AVERAGE IS ";A
```

Perhaps the difficulty is still not apparent. Another technique we can use is the **TRACE** statement in BASIC.

The Trace

We will illustrate use of the trace on the above program. Let's turn on the trace function by typing **TRACE** . The computer will respond by typing]. Now type **RUN**. The computer will run our program and print out the line numbers of all executed instructions. Here is what our display looks like:

```
TRACE
[
RUN
#20 THIS PROGRAM COMPUTES AN AVERAGE
#30 TYPE IN NUMBERS ONE AT A TIME
#40 TYPE 0 WHEN FINISHED
#50 #60 ?1
#70 #80 #90 #50 #60 ?2
#70 #80 #90 #50 #60 ?0
#70 #100 #105 3 3
#110 THE AVERAGE IS 1
```

The numbers preceded by # indicate the line numbers executed. The values not preceded by # represent the program output resulting from execution. We see that line number 50 is executed three times, thus explaining the value of I = 3, although we really only have two valid input items; the last is only a flag datum. We can correct this by decreasing the value of I before we use it to compute the average. (This is not the only way that the correction can be

made. Can you think of another?) To do this we will change statement 100 to read

```
100 A=S/(I-1)
```

Now if we rerun the program, we see that the average comes out correctly.

To remove the trace simply type in

```
NOTRACE
```

Now let's remove the extra print statement which we inserted for debugging purposes (line 105) and rerun the program with another set of test data. The program currently looks like this:

```
10 HOME
20 PRINT "THIS PROGRAM COMPUTES AN AVERAGE"
30 PRINT "TYPE IN NUMBERS ONE AT A TIME"
40 PRINT "TYPE 0 WHEN FINISHED"
50 I=I + 1
60 INPUT N
70 IF N=0 THEN 100
80 S=S + N
90 GOTO 50
100 A=S/(I-1)
110 PRINT "THE AVERAGE IS ";A
```

Let's find the average of 2, 4, and 6 which we know should equal 4.

```
RUN
THIS PROGRAM COMPUTES AN AVERAGE
TYPE IN NUMBERS ONE AT A TIME
TYPE 0 WHEN FINISHED
?2
?4
?6
?0
THE AVERAGE IS 4
```

This time the answer is correct, and we have reason to believe that it will run correctly on any set of data.

Exercises (answers on page 317)

1. Use the error messages to debug the following program to calculate $(1^2+2^2+\ldots+50^2)(1^3+2^3+\ldots+20^3)$.

```
10 S="0"
20 FOR J=1 TO 100
30 S=S + J(2
40 NEXT K
50 LET T=0
60 FOR J=1 TO 30
70 LET T=T +J^3
```

```
80 NXT T
90 NEXT T
100 LET A=ST
110 PRINT THE ANSWER IS, A
120 END
```

2. Use the trace function to debug the following program to determine the smallest integer N for which N^2 is larger than 175263.

```
10 LET N=0
20 IF N^2 > 175263 THEN 100
30 PRINT "THE FIRST N EQUALS"
100 N=N+1
110 GOTO 10
200 END
```

5.8 Appendix—Some Common Error Messages

? SYNTAX ERROR. There is an unclear instruction (misspelled?), mismatched parentheses, incorrect punctuation, illegal character, or illegal variable name in the program.

? UNDEF'D STATEMENT ERROR. The program uses a line number which does not correspond to an instruction. This can easily occur from deleting lines which are mentioned elsewhere. It can also occur when testing a portion of a program which refers to a line not yet written.

? UNDEF'D FUNCTION ERROR. The program uses a function which has not been defined in a DEF statement.

? OVERFLOW ERROR. A number too large for the computer.

? DIVISION BY ZERO ERROR. Attempting to divide by 0. This may be a hard error to spot. The computer will round to 0 any number smaller than the minimum allowed. Use of such a number in subsequent calculations could result in division by 0.

? ILLEGAL QUANTITY ERROR. (For the mathematically-minded) Attempting to evaluate a function outside of its mathematically defined range. For example, the square root function is defined only for non-negative numbers, the logarithm function only for positive numbers, and the arctangent only for numbers between -1 and 1. Any attempt to evaluate a function at a value outside these respective ranges will result in an ILLEGAL QUANTITY error.

? BAD SUBSCRIPT ERROR. Attempting to use an array with one or more subscripts outside the range allowed by the appropriate DIM statement.

? STRING TOO LONG ERROR. Attempting to specify a string containing more than 255 characters.

? OUT OF MEMORY ERROR. Your program will not fit into the computer's memory. This could result from large arrays, too many program steps, or a combination of the two.

? FORMULA TOO COMPLEX ERROR. Due to the internal processing of your formula, your formula resulted in a string expression that was too long or complex. This error can be corrected by breaking the string expression into a series of simpler expressions.

? TYPE MISMATCH ERROR. Attempting to assign a string constant as the value of a numeric variable or a numeric constant to a string variable.

? REDIM'D ARRAY ERROR. Attempting to DIMension an array which has already been dimensioned. Note that once you refer to an array within a program, even if you don't specify the dimensions, the computer will regard it as being dimensioned at 10.

? NEXT WITHOUT FOR ERROR. A NEXT statement which does not correspond to a FOR statement.

? RETURN WITHOUT GOSUB ERROR. A RETURN statement encountered while not in a subroutine (we will discuss subroutines in the next chapter).

? OUT OF DATA ERROR. Attempting to read data which isn't there. This can occur in reading data from DATA statements, cassettes, or diskettes.

? CAN'T CONTINUE ERROR. Attempting to give a CONT command after the program has ENDed, or before the program has been RUN (such as after an EDIT session).

6
Organizing a Program

In Chapter Four we discussed how to structure a problem solution. We used this structuring technique quite a bit in developing the application programs in Chapter Five.

In this chapter we will talk about giving some structure to the program itself. The chapter is divided into three sections. The first discusses a way in which you can define your own functions similar to the SQR, INT, and RND functions you have already seen. Then we will discuss breaking the program up into subprograms. We have already done this with the problem solution and used this to develop the program, but in this section we will formally define these pieces as separate parts of the program. Finally, in the last section, we will give you some tips to make your programs run a little more quickly. While these are not usually too important for someone who is not developing software professionally, there are occasions when they could be useful to you.

6.1 Defining Your Own Functions

We have already discovered many functions which are part of the Applesoft BASIC language in Section 3.8.

In mathematics, functions are usually defined by specifying one or more formulas. For instance, these are formulas which define the functions of the variable x, f(x), g(x), and h(x):

$f(x) = (x^2 - 1)^{1/2}$
$g(x) = 3x^2 - 5x - 15$
$h(x) = 1/(x-1)$

Each function is named by a letter, namely f, g, and h, respectively.

Applesoft BASIC allows you to define functions like these and use them by name throughout your program. To define a function you use the DEF FN instruction. This instruction is used before the first use of the function in the program.

For example, to define the function f(x) above, we could use the instruction

```
10 DEF FNF(X)=(X^2 - 1)^(1/2)
```

To define the function g(x) above, we use the instruction

```
20 DEF FNG(X)=3*X^2 - 5*X - 15
```

Note that in each case we use a letter (in these examples, F and G) to identify the function. The variable name is preceded by the letters FN to tell the computer that this is a function and not just a numeric variable.

Suppose we wish to calculate the value of the function F for X = 12.5. Once the function has been defined, this calculation may be described to the computer as FNF(12.5). Such calculations may be used throughout the program and save the effort of retyping the formula for the function in each instance.

You may use any valid variable name as a function name. For example, you may define a function RATE by the statement

```
10 DEF FNRATE(X) = ...
```

Remember that just as is true for any variable name used in Applesoft, only the first two letters of the variable are used to identify the function. Any other variable name beginning with RA would refer to the same function as the one above.

In defining a function, you may use other functions. For example, if FNF(X) and FNG(X) are as defined above, then we may define their product by the instruction

```
30 DEF FNC(X)=FNF(X)*FNG(X)
```

One place we would find a function very useful is in the use of the rounding expression. For example, to round the variable X to two decimal places, we use the statement

```
20 X=INT(X * 100 + .05)/100
```

If we defined this as a function, we could save a great deal of typing time. Let's call the function RD for round. The function definition would be

```
10 DEF FNRD(X)=INT(X * 100 + .05)/100
```

If we wanted to print out the value of X rounded to two places, we could just use

```
20 PRINT FNRD(X)
```

Review the investment analysis system developed in Section 5.4. Note that many times we could have made use of the function definition in that program.

Exercises (answers on page 317)

Write instructions to define the following functions.

1. $x^2 - 5x$
2. $1/x - 3x$
3. $5\exp(-2x)$
4. $x \log(x/2)$
5. $\tan x / x$
6. $\cos(2x) + 1$
7. Write a program which tabulates the value of the function in Exercise 3 for x = 0, .5, 1.0, 1.5, ... 10.0.

6.2 Subroutines

The function definition capability is very useful, but it is limited because the functions which can be defined are limited to one line only. In writing programs it is often necessary to use the same sequence of instructions more than once. It may not be convenient (or even feasible) to retype the set of instructions each time it is needed. Fortunately, Applesoft BASIC offers a convenient alternative: the *subroutine*.

A *subroutine* is a program which is incorporated within another, larger program. The subroutine may be used any number of times by the larger program. Often, the lines corresponding to a subroutine are isolated toward the end or beginning of the larger program. This arrangement is illustrated in Figure 6-1. The arrow to the subroutine indicates the point in the larger program at which the subroutine is used. The arrow pointing away from the subroutine indicates that, after completion of the subroutine, execution of the main program resumes at the point at which it was interrupted.

Subroutines are handled with the pair of instructions **GOSUB** and **RETURN**. The statement

```
100 GOSUB 1000
```

sends the computer to the subroutine which begins at line 1000. The computer starts at line 1000 and carries out statements in order. When a

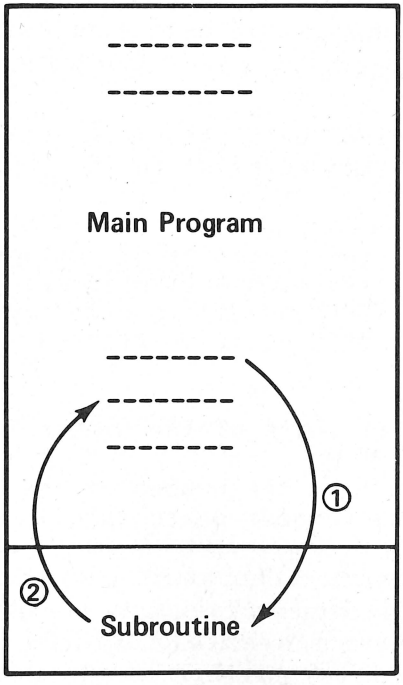


Figure 6-1. A subroutine.

RETURN statement is reached, the computer goes back to the main program, starting at the first line after 100.

It is important to recognize the difference between a **GOSUB** and a **GOTO**. In a **GOTO** the computer branches to the line number given and continues from that point until it either reaches the end of the program or hits another control statement which sends it somewhere else. With the **GOSUB**, on the other hand, the computer remembers the point at which it left the main program and automatically returns to that point when it reaches the **RETURN** statement.

The next example illustrates the use of subroutines.

Example 1. Here are production figures for six assembly lines of a factory for January 1982 and January 1983. Calculate the percentage increase (or decrease) for each assembly line and determine the assembly line with the largest percentage increase.

Assembly Line	January 1982	January 1983
1	235,485	239,671
2	298,478	301,485
3	328,946	322,356
4	315,495	318,458
5	198,487	207,109
6	204,586	221,853

Solution. Plan the program in terms of functions and subroutines. Store the January 1982 data in the array A(J) and the January 1983 data in the array B(J), J = 1,2,3,4,5,6. Our first step will be to read the data into the arrays from **DATA** statements.

```
initialize:  get 1982 data (store in A(J))
             get 1983 data (store in B(J))

loop for each assembly line
     percentage:  calculate the percentage increase
                  (decrease) as B(J)-A(J) divided by A(J)
                  and the entire quantity multiplied by
                  100, store in C(J)
endloop

largest:  determine which of the numbers C(J) is the
          largest
          and let K be the number of the assembly line
          with the largest percentage increase
```

The **percentage** block can easily be written in one line and thus is appropriate for a function. The **largest** block can be placed in a subroutine.

Let's call the function for percentage PR. To define this function we use

```
5 DEF FNPR(J)=( (B(J)-A(J))/A(J) ) * 100
```

For the **largest** block we need several statements. The solution procedure is

```
largest:
     set maximum equal to first line (MX=C(1) and K=1)
     loop for lines 2 through 6
         if increase for line is greater than maximum,
              change maximum
     endloop
```

This will require several lines to implement in BASIC, thus we need to write a subroutine like this:

```
2000 K=1: MX=C(1)
2010 FOR J=2 TO 6
2020   IF C(J) > MX THEN K=J: MX=C(J)
2030 NEXT J
2040 RETURN
```

The entire program is:

```
5 DEF FNPR(J)=( (B(J)-A(J))/A(J) ) * 100
10 DIM A(6),B(6),C(6)
20 DATA 235485, 239671, 298478, 301485
30 DATA 328946, 322356, 315495, 318458
40 DATA 198487, 207109, 204586, 221853
50 FOR J=1 TO 6: REM READ ARRAYS
60 READ A(J), B(J)
70 NEXT J
80 FOR J=1 TO 6: REM COMPUTE PERCENT INCREASES
90    C(J)=FNPR(J)
100 NEXT J
200 PRINT "ASSEMBLY LINE","PERCENT INCREASE"
210 FOR J=1 TO 6: REM DISPLAY PERCENT INCREASES
220    PRINT J, C(J)
230 NEXT J
300 GOSUB 2000: REM DETERMINE LARGEST PERCENT INCREASE
310 PRINT "ASSEMBLY LINE",K,"IS THE WINNER"
400 END
2000 K=1: MX=C(1)
2010 FOR J=2 TO 6
2020   IF C(J) > MX THEN K=J: MX=C(J)
2030 NEXT J
2040 RETURN
```

The point here is that the use of functions and subroutines allowed us to organize the program and to plan it in much the same way that we set up our solution procedure so that programming may be done in small steps. While we could write the program without the use of subroutines, their use allows us to write the main program, the functions, and the subroutines without worrying about the program in its entirety.

TEST YOUR UNDERSTANDING 1 (answer on page 158)

Consider the following program.

```
10 GOSUB 500
20 A=5
30 END
500 B=7
510 GOSUB 600
600 C=9
700 RETURN
800 END
```

What is the line executed after line 700?

Here is a useful variation of the **GOSUB** statement. Suppose that you wish to use

1. the subroutine beginning at line 100 if the value of J is 1,
2. the subroutine beginning at line 500 if the value of J is 2, and
3. the subroutine beginning at line 1000 if the value of J is 3.

This complex set of decisions can be requested by a simple statement of the form

```
10 ON J GOSUB 100,500,1000
```

If the value of J is 1, the program goes to line 100; if the value of J is 2, the program goes to line 500; if the value of J is 3, the program goes to line 1000. If the value of J is not an integer, then any fractional part will be ignored. For example, if the value of J is 5.5, then the value 5 will be used. After dropping the fractional part, the value must be between 0 and 255 or an error will occur. If the value of J does not correspond to a given subroutine (that is, if J is 0 or more than 3 in the above example), then the instruction will be ignored.

Instead of J, we could use any valid expression, such as J^2 + 3 or 3*J - 2. The expression will be evaluated and all of the above rules apply. This is very similar in operation to the ON...GOTO statement we discussed previously except that control will now return to the statement immediately following the ON...GOSUB.

Example 2. Modify the roulette program of Example 5, Section 5.3, so it allows bets on EVEN and ODD. A one dollar bet on either of these pays one dollar in winnings.

Solution. Our program will now allow three different bets: on a number, on EVEN, or on ODD. Let us design subroutines, corresponding to each of these bets, which determine whether player J wins or loses. For each subroutine let X be the number (1-38) which results from spinning the wheel. In the preceding program a bet by player J was described by two numbers: B(J)

equaled the number bet and C(J) equaled the amount bet. Now let us add a third number to describe a bet. Let D(J) equal 1 if J bets on a number, 2 if J bets on EVEN, and 3 if J bets on ODD. In case D(J) is 2 or 3, we will again let C(J) equal the amount bet, but B(J) will be 0. The new solution procedure represents an addition to the procedure in Section 5.3. Here it is.

```
      initialize: get the number of players (N)
loop  for each player
      input chips: get number of chips for each player
endloop
loop  for each game
      loop for each player
           bets: get the type of bet (number,even,odd),
                 if number, get number to be played,
                 get amount bet
      endloop
      spin wheel: generate random integer between 1 and 38
      (X)
      loop for each player
           determine wins and losses:
                number bet:
                     if number bet=X then player wins 32
                          times amount bet, add to account
                     if not, then subtract amount bet from
                          account
                even bet:
                     if number is even then player wins the
                          amount of the bet, add to account
                     if not, then subtract amount bet from
                          account
                odd bet:
                     if number is odd then player wins the
                          amount of the bet, add to account
                     if not, then subtract amount bet from
                          account
      endloop
      loop for each player
           output: player number and account amount
      endloop
end game loop
```

The three possible ways to win correspond to the three subroutines in our program. Let's work on the subroutines first.

The subroutine for determining the winners of bets on numbers can be obtained by making small modifications to the corresponding portion of our previous program as follows:

```
1000 IF B(J)=X THEN 1100
1010 PRINT "PLAYER ";J;" LOSES"
1020 A(J)=A(J) - C(J)
```

```
1030 RETURN
1100 PRINT "PLAYER ";J;" WINS"; 32*C(J);"DOLLARS"
1110 A(J)=A(J) + 32*C(J)
1120 RETURN
```

Here is the subroutine corresponding to the bet EVEN.

```
2000 FOR K=1 TO 19
2010 IF X=2*K THEN 2100
2020 NEXT K
2030 PRINT "PLAYER ";J;" LOSES"
2040 A(J)=A(J) - C(J)
2050 RETURN
2100 PRINT "PLAYER ";J;" WINS ";C(J);" DOLLARS"
2110 A(J)=A(J) + C(J)
2120 RETURN
```

Finally, here is the subroutine corresponding to the bet ODD.

```
3000 FOR K=1 TO 19
3005 IF X=2*K THEN 3030
3010 NEXT K
3020 GOTO 3100
3030 PRINT "PLAYER ";J;" LOSES"
3040 A(J)=A(J) - C(J)
3050 RETURN
3100 PRINT "PLAYER ";J;" WINS ";C(J);" DOLLARS"
3110 A(J)=A(J) + C(J)
3120 RETURN
```

Now we are ready to add the subroutines to the main portion of the program, which is almost the same as before. The only essential alteration is that we must now determine, for each player, which bet was placed. Here is the program.

```
10 INPUT "NUMBER OF PLAYERS";N
20 DIM A(N),B(N),C(N)
25 REM LINES 30-60 ALLOW PLAYERS TO PURCHASE CHIPS
30 FOR J=1 TO N : REM FOR EACH OF THE PLAYERS
40   PRINT "PLAYER "; J
50   INPUT "HOW MANY CHIPS"; A(J)
60 NEXT J
100 PRINT "LADIES AND GENTLEMEN!"
105 PRINT "PLACE YOUR BETS PLEASE!"
110 FOR J=1 TO N : REM FOR EACH OF THE PLAYERS
120  PRINT "PLAYER "; J
130  PRINT "BET TYPE:1=NUMBER BET, 2=EVEN, 3=ODD"
140  B(J)=0: INPUT "BET TYPE (1,2 OR 3) ";D(J)
150  ON D(J) GOTO 160,170,170
160  INPUT "NUMBER ON WHICH TO BET? ";B(J)
170  INPUT "AMOUNT "; C(J)
180 NEXT J
```

```
200 X=INT(38*RND(1) + 1): REM SPIN THE WHEEL
210 REM LINES 210-300 DISPLAY THE WINNING NUMBER
220 PRINT "THE WINNER IS NUMBER "; X
300 REM: LINES 310-330 DETERMINE WINNINGS AND LOSSES
310 FOR J=1 TO N : REM FOR EACH PLAYER
320  ON D(J) GOSUB 1000,2000,3000
330 NEXT J
400 PRINT "PLAYER BANKROLLS"
410 PRINT
420 PRINT "PLAYER", "CHIPS"
430 FOR J=1 TO N
440  PRINT J,A(J)
450 NEXT J
460 INPUT "DO YOU WISH TO PLAY ANOTHER ROLL(Y/N)";R$
470 HOME
480 IF R$="Y" THEN 100
490 PRINT "THE CASINO IS CLOSED. SORRY!"
500 END
1000 IF B(J)=X THEN 1100
1010 PRINT "PLAYER ";J;" LOSES"
1020 A(J)=A(J) - C(J)
1030 RETURN
1100 PRINT "PLAYER ";J;" WINS "; 32*C(J);" DOLLARS"
1110 A(J)=A(J) + 32*C(J)
1120 RETURN
2000 FOR K=1 TO 19
2010 IF X=2*K THEN 2100
2020 NEXT K
2030 PRINT "PLAYER ";J;" LOSES"
2040 A(J)=A(J) - C(J)
2050 RETURN
2100 PRINT "PLAYER ";J;" WINS ";C(J);" DOLLARS"
2110 A(J)=A(J) + C(J)
2120 RETURN
3000 FOR K=1 TO 19
3005 IF X=2*K THEN 3030
3010 NEXT K
3020 GOTO 3100
3030 PRINT "PLAYER ";J;" LOSES"
3040 A(J)=A(J) - C(J)
3050 RETURN
3100 PRINT "PLAYER ";J;" WINS ";C(J);" DOLLARS"
3110 A(J)=A(J) + C(J)
3120 RETURN
```

Note how the subroutines help organize our programming. Each subroutine is easy to write. Each is a small task and you will have less to think about than when considering the entire program. It is advisable to break a long program into a number of subroutines. Not only is it easier to write but it is much easier to check the program and locate errors since subroutines may be individually tested.

Exercises (answers on page 317)

1. Write a subroutine computing the value of 5*J^2 - 3*J. Incorporate it in a program evaluating the given expression at J = .1, .2, .3, .4, .5.
2. Write the subroutine of Example 1 on page 152 which computes the percentage rates of change for each assembly line. Use the formula

<% rate of change> = 100 x (<Jan.1983 Prod.>-<Jan.1982 Prod.>)/<Jan.1982 Prod.>

3. Write a subroutine for Example 1 determining the largest percentage increase. Hint: Let the variable M hold the largest number among C(1),...,C(6). At the start, set M to C(1). Repeatedly compare M to C(2),C(3),...,C(6). After each comparison, replace M by the larger of M and what is used for comparison. At the end of 5 comparisons, M will contain the largest among C(1),C(2),...,C(6). You may then determine to which assembly line that value of M belongs. Is the final value of M equal to C(1), C(2), C(3), C(4), C(5), or C(6)? For example, if M equals C(5), then assembly line 5 has the largest percentage increase.
4. Run the program of Example 1 with the subroutines in place.
5. Modify the roulette program of Example 2 to allow the following bets: first 12 (1-12), second 12 (13-24), and third 12 (25-36). A winning one-dollar bet of this type pays two dollars.
6. Rewrite the Social Security Number input program of Exercise 1, Section 5.5 using a subroutine.

ANSWERS TO TEST YOUR UNDERSTANDING 1

The lines will be executed in the following order:

10, 500, 510, 600, 700, 600, 700, 20, 30

6.3 Some Programming Efficiencies

In this section we will discuss some techniques you can use to increase the speed with which your programs execute. Most of these techniques can become habit after a while. Just remember: the ability to understand what you are doing in your program, or to enable someone else to understand your program, is more important for the non-professional programmer than having a program which runs a second or two faster. Clarity, for yourself or someone else, often conflicts with efficiency. **Do not** use these techniques if they make it more difficult for you to trace through your program.

Any program will execute more rapidly by putting several statements on the same line. In addition to this, we can divide these programming tips into three general categories: arithmetic operations, branching, and looping.

Arithmetic Operations

You write your programs in BASIC. The computer, however, must translate your programs into its own machine language before it can actually execute a statement. The usual approach to this translation for BASIC programs is via an *interpreter.*

The interpreter translates the statements into internal code one at a time as it executes them. For an interpreter this is frequently a two-step process. First is a partial translation; then, at execution, the translation is completed.

When you first use a variable, the partial translation for this variable occurs. This half-way step is performed each time you redefine the variable; each time it appears on the left side of an equation. When this variable is subsequently used on the right-hand side in an arithmetic operation, only the second part of the translation need be done because the computer recognizes the variable name.

In the case of a constant the computer must make the entire translation each time it encounters the constant value. Thus a statement set like

```
10 FOR I=1 to 10
20   B=2.34 * A(I)
30 NEXT I
```

requires that the 2.34 be translated into machine language each time the program goes through the loop. Some time can be saved by defining 2.34 as a variable first. A more efficient program would read

```
5 X=2.34
10 FOR I=1 to 10
20   B=X * A(I)
30 NEXT I
```

Some operations require more time than others. Thus, writing your arithmetic expressions in certain ways can speed up execution. For example, use:

```
X * X      instead of  X^2
X + X      instead of  2 * X
X * .25    instead of  X/4
```

Again, remember that being able to understand your program is usually more important than saving a little time.

Finally, don't make the computer evaluate the same expression more than once. If you have a set of statements like

```
10 L(I + 1)=S + L(I)
20 B=L(I + 1)/(A + B)
```

you can save a little time by writing

```
5 X=I + 1
10 L(X)=S + L(I)
20 B=L(X)/(A + B)
```

Similarly, you can simplify expressions to avoid repeating operations. For example:

```
10 X=Y + 2 * Y^2
```

requires one exponentiation, one multiplication, and one addition. The same expression could be written as:

```
20 X=Y * (1 + 2 * Y)
```

which requires only two multiplications and one addition.

Using these techniques may require additional lines of code. In some cases memory space may be more critical than speed. In these instances you must make a trade-off between space and time.

Branching

The execution of an **IF...THEN** statement is generally time consuming. Sometimes, you can arrange the statements in your program so that as few IFs as possible are executed before a branch occurs.

For example, suppose you were writing a program which would calculate student grades and then assign a grade of A if the numeric average were 90 or greater, a B if the grade was greater than or equal to 80 but less than 90, a C for grades greater than or equal to 65 but less than 80, and an F for grades below 65. If you knew the typical grade distribution led to most students receiving B's with C's the next most frequent, followed by A's and then F's, you could arrange your program to reflect this:

```
1000 IF AV >=80 AND AV < 90 THEN G$="B":GOTO 1100
1010 IF AV >=65 AND AV < 80 THEN G$="C":GOTO 1100
1020 IF AV >=90 THEN G$="A":GOTO 1100
1030 G$="F"
1100 PRINT "THE GRADE FOR THIS STUDENT IS ";G$
```

Most students would require only one IF...THEN. If the student did not qualify in any of the first three categories, then we know that the grade is an "F", so there is no need to ask a question in this case. Of course, you will not always know the frequency of occurrence, but in general, you should try to reduce the number of conditions which have to be evaluated by arranging the conditional branches in order of likelihood and then branching out of the IF...THEN sequence.

When you use a GOTO or GOSUB statement in your Applesoft program, the computer searches for the requested line number beginning at the *top* of your program. This means that if there is some routine to which you branch frequently, you can save some time by putting this routine toward the start of your program. If you look at professionally developed programs for the Apple II Plus or Apple IIe, you will often see the subroutines placed at the start of the program for this reason.

Looping

When using loops, the most important thing to remember is not to place operations which do not need to be repeated inside the loop. For example:

```
10 FOR I=1 TO 30
20    X=A + B
30    Y=X^I
40 NEXT I
```

can be written as

```
10 X=A + B
20 FOR I=1 TO 30
30    Y=X^I
40 NEXT I
```

This saves evaluating A + B thirty times.

Choosing appropriate limits for the loop can save time, too. For example, if we have data for the years 1975 through 1982 stored in an array called D in locations 1 through 8, we could call this data by using

```
10 FOR I=1975 TO 1982
20 PRINT D(I-1974)
30 NEXT I
```

or we could simply set the loop limits from 1 TO 8

```
10 FOR I=1 TO 8
20 PRINT D(I)
30 NEXT I
```

Another thing to watch out for in using loops is unnecessary repetition of the FOR...NEXT. For example

```
10 DIM T(20)
20 FOR I=1 TO 20
30   READ T(I)
40   DATA 23,45,.....
50 NEXT I
60 FOR I=1 TO 20
70   S=S + T(I)
80 NEXT I
90 PRINT "THE TOTAL IS ";S
100 END
```

could be done in one loop.

```
10 DIM T(20)
20 FOR I=1 TO 20
30   READ T(I)
40   DATA 23,45,.....
70   S=S + T(I)
80 NEXT I
90 PRINT "THE TOTAL IS ";S
100 END
```

This saves a good deal of time because whenever the NEXT statement is encountered, the computer must evaluate to see whether the end of the loop is encountered. By combining these two loops into one, we eliminate the equivalent of 20 IF statements.

Another time saver with respect to FOR...NEXT loops is the elimination of the index variable name after the word NEXT. Consider the following statements:

```
10 FOR X=1 TO 50000
20 PRINT X
30 NEXT X
```

You could accomplish the same action by simply using the word NEXT as in the following statements:

```
10 FOR X=1 TO 50000
20 PRINT X
30 NEXT
```

Notice the difference in line 30. Starting a FOR...NEXT loop causes the computer to set up a counter for the index variable which it increments and tests each time through the loop. When the computer encounters a NEXT statement, it automatically tests the value of the index variable for the last uncompleted loop. Using this technique, though, may cause some problems if you are not careful. If you branch out of a loop using a GOTO without bringing the loop to a normal completion, the computer will continue to keep track of that index variable. A NEXT statement which does not specify a variable may, therefore, return you to the loop from which you branched instead of the one you intended it to execute.

Remember that none of these efficiencies make sense for the non-professional programmer if including them makes the program more difficult to understand. Doing so will lead to more logic errors and more debugging time. Your purpose is to develop usable programs even if they take a little longer to execute.

7
Using Your Computer as a File Cabinet

In the previous chapters we learned the fundamentals for programming the Apple II Computer. (Remember that in this book when we refer to the Apple II computer, we mean the Apple II Plus and the Apple IIe.)

In this chapter we will discuss the use of the disk for writing and reading data files. In Section 7.1 we will look at the difference between program files and data files. In the remainder of the chapter, we will discuss the different types of data files and provide some application programs.

7.1 What Are Data Files?

Computer programs as used in business and industry usually refer to files of information which are stored as some secondary-storage medium and brought into RAM when the computer executes the instructions in your programs. For example, a personnel department would keep a file of personal data on each employee containing name, age, address, social security number, date employed, position, salary, and so forth. A warehouse would maintain an inventory file for each product with the following information: product name , supplier, current inventory, units sold in the last reporting period, date of last shipment, size of last shipment, and units sold in the last 12 months. Files like these are called **data files**. They could contain hundreds, thousands, or even hundreds of thousands of entries.

Data files are usually stored using some mass-storage medium. In the case of the Apple II Computer, this storage medium is a disk or tape. Data files are to be distinguished from **program files** which result from saving programs. Program files and data files exist side by side on the disk.

There are two main types of data files that differ in the way data is stored and accessed. The first type is called a **sequential access** file. In this type of data file, pieces of data are stored one after the other in sequence. In order to access a specific piece of data, you need to read all the preceding data items first. Because of this limitation, sequential files are useful only if the programs using that data require that the records be accessed one after the other. A teacher may use a sequential access file to store information about grades on the assumption that when the file has to be accessed, it will

usually be desirable to treat each student, for example, to add a grade for each student or to calculate a final average for each.

The second type of file is called a **random access** file. With this type of file, it is possible to read or write any piece of information anywhere in the file without having to read all the preceding data. The ability to access any desired piece of data at random gives the file its name. These are very useful files for many applications because of their flexibility. A file on inventory which kept track of the number of items in stock, whether an order was currently outstanding, and the size of the order would probably best be stored as a random access file. That way it would be possible to look up the current status on any individual part or change the information associated with that part without having to affect any other data.

Your Apple II Computer can read data from a data file while a program is running, so the program can use the data. The file may be used within the program.

For example, a personnel department might have a program which (1) enters changes in the personnel data file and (2) displays requested information about a given employee. To perform task (1), the program would read the personnel data file, alter the relevant items and rewrite the file into mass storage. To perform task (2), the program would read the data file, search for the requested information and display it on the screen or printer. In effect, your computer is serving as a convenient file cabinet for the storage of data. Moreover, the programming capability of the computer allows you to easily "shuffle through" the data for a specific piece of information.

Using DOS Commands in BASIC Programs

In Section 2.6 we introduced some DOS commands like CATALOG. In this chapter we will learn the DOS commands required to access data files stored on disk.

To allow your Apple II to access the disk to read and write data files or indeed to perform any disk function within a BASIC program, we need to do more than just state the DOS command. For example, we can type the word

```
CATALOG
```

and, in immediate execution mode, the computer would immediately display a list of the files on the diskette in the current disk drive. However, if we wrote a BASIC program and tried to use this statement directly, all we would obtain would be a syntax error. For example, consider the following:

```
10 PRINT "HERE IS A LIST OF AVAILABLE PROGRAMS"
20 CATALOG
30 PRINT "TO RUN A PROGRAM TYPE IN THE WORD RUN"
40 PRINT "AND THE NAME OF THE PROGRAM"
50 END
```

If you tried to run this program you would obtain a syntax error at line 20. How can we cause the computer to list a catalog while executing the BASIC

program? To do this we need to tell the computer that we are sending a DOS command.

To tell the computer that we are sending a DOS command, we need to know a little about how the computer stores string values. Each key or key combination that you can strike enters the computer as a numeric code. For example, the numeric code for the letter A is 65, for the RETURN key is 13, and for the digit 1 is 49. These codes are standard to microcomputers and are part of the **American Standard Code for Information Interchange (ASCII)**. For this reason these numeric codes are referred to as ASCII (pronounced asskey) codes. CHR$(X) is a built-in BASIC function like the ones we discussed in Chapter Three. It instructs the computer to use the character whose ASCII code is given in parentheses.

To understand the use of this function, try typing in the following:

```
PRINT CHR$(65)
PRINT CHR$(49)
```

The first statement should have resulted in an A being printed, the second the number 1. A list of ASCII codes is given in the appendix to this chapter.

Now let's get back to issuing DOS commands within the BASIC program. To tell the computer that we want to issue a DOS command, we must send it the control sequence, CONTROL-D, which has an ASCII code of 4. We send the computer this control sequence followed by the DOS command inside of a BASIC PRINT statement. If we merely typed in the CONTROL-D sequence in the PRINT statement, the statement would look like this:

```
20 PRINT "CATALOG"
```

As you can see, the CONTROL-D does not print out. This can frequently lead to errors in the interpretation of a statement. To avoid these potential problems, we take advantage of the fact that the ASCII code for CONTROL-D is 4. Thus, instead of typing in CONTROL-D, we type in CHR$(4). Using the CHR$ function to represent the CONTROL-D sequence, the equivalent but less misleading statement to the above request to display the catalog is

```
20 PRINT CHR$(4);"CATALOG"
```

Try substituting this statement into the program listed above. Now you should see the disk catalog printed out as part of the program execution. When this control sequence is used frequently, it is sometimes set equal to a string variable early in the program to save some entry time; for example, the statement

```
10 D$=CHR$(4)
```

might be used and then subsequent referrals to the CONTROL-D sequence would use D$. We will need to use this control sequence as we discuss setting up, writing, and reading data files in this chapter.

You can also turn your printer on and off using commands in your program like

`PRINT D$;"PR#1"` to turn it on if the interface board is in slot 1, and

`PRINT D$;"PR#0"` to turn it off

You may also need to send additional control characters to the printer. If so, they too can be sent using a similar statement in your BASIC program.

Using the Trace Function with DOS Commands

In an earlier chapter we discussed using the TRACE command to help debug a program. In order to use the TRACE function when we have DOS commands in our program, we must define D$ a little differently. When the TRACE function is in effect, a carriage return is not generated at the end of the execution of each TRACE operation. However, when a DOS command is executed, the computer expects to find the PRINT D$ at the beginning of a new line. We can correct the difficulty by defining D$ as

```
D$=CHR$(13) + CHR$(4)
```

The 13 is the ASCII code for a RETURN. Thus, we will force a carriage return before executing the DOS command. If you plan to use the TRACE command in your program, you should bear this in mind.

7.2 Using Sequential Files

The following example will give you a better idea of how a sequential file is organized within mass storage. Suppose a teacher stores grades in a data file. For each student in the class there are currently three exam grades. A typical entry in the data file would contain the following data items:

student name, exam grade #1, exam grade #2, exam grade #3

In the data file, the data items are organized in sequence. The beginning of the above data file might look like this:

'John Smith', 98, 87, 93, 'Mary Young',
99, 78, 87, 'Sally Ronson', 48, 63, 72...

The data file consists of a sequence of either string constants (the names) or numeric constants (the grades), with the various data items arranged in a particular pattern (name followed by three grades).

This particular arrangement is designed so the file may be read and understood. If we read the data items above, we know in advance that the data items are in groups of four with the first one a name and the next three the corresponding grades. In order for the computer to know where one data item ends and another begins, you must separate consecutive data items by characters called **delimiters**. Commas and the RETURN are examples of commonly used delimiters.

In order to either read or write a data file on diskette, it is necessary to **OPEN** the file. When opening a file, you tell DOS whether you plan to read the file or write the file. For example, to write a sequential file with the name COURSE.211 on disk drive 2, we would use a sequence of instructions like this:

```
100 PRINT CHR$(4); "OPEN COURSE.211, D2"
110 PRINT CHR$(4); "WRITE COURSE.211, D2"
```

Note that these instructions do not actually write any data into the file. They merely prepare the file for writing.

Suppose that we now wish to enter the data for John Smith into the file

JOHN SMITH 98 87 76

We could use the following instruction:

```
200 PRINT "JOHN SMITH,";98;",";87;",";76
```

Note we are instructing the computer to print out the name and three grades so that there is a comma between each two pieces of data. This is because we need to have delimiters separating the data, and these delimiters are not ordinarily printed. We could also have used separate print statements for each data item. In this way, a return would automatically have been placed at the end of each item and would have served as a delimiter. Then we wouldn't need the commas.

Once the **WRITE** command has been given, all subsequent PRINT statements will cause data to be written into the file indicated. Any statement which would ordinarily cause output to appear on the screen will cause that same output to be written into the open file on disk. Thus

```
LIST 10,90
```

would ordinarily cause lines 10 through 90 to be printed on the screen. If you have opened a file for writing, when you issue this command in your program, those lines will be written onto the disk file instead of the screen.

TEST YOUR UNDERSTANDING 1 (answer on page 174)

What is the name of the data file used in the statements below? What will be written into this file as a result of the following statements?

```
10 PRINT "HELLO"
20 PRINT "GOODBYE"
30 PRINT CHR$(4);"OPEN TRIAL"
40 PRINT CHR$(4);"WRITE TRIAL"
50 PRINT "THIS IS A TEST"
60 LIST 10,20
```

When you are finished writing a file, you must close it with a **CLOSE** instruction. To close the student file we use the instruction

```
300 PRINT CHR$(4); "CLOSE COURSE.211,D2"
```

Example 1. Create a data file consisting of names, addresses, and telephone numbers from your personal telephone directory. Assume that you will type the addresses into the computer and will tell the computer when the last address has been typed.

Solution. We use **INPUT** statements to enter the various data. Let A$ denote the name of the current person, B$ the street address, C$ the city , D$ the state, E$ the zip code, and F$ the telephone number. For each entry, there is an **INPUT** statement corresponding to each of these variables. The data is put into arrays as it is inputted. The program then writes the data to the diskette. Here is the program.

```
10 INPUT "NUMBER OF ENTRIES IN DIRECTORY? ";N
20  DIM A$(N),B$(N),C$(N),D$(N),E$(N),F$(N)
30 FOR J=1 TO N
110 INPUT "NAME"; A$(J)
120 INPUT "STREET ADDRESS"; B$(J)
130 INPUT "CITY"; C$(J)
140 INPUT "STATE"; D$(J)
150 INPUT "ZIP CODE"; E$(J)
160 INPUT "TELEPHONE"; F$(J)
170 NEXT J
200 PRINT CHR$(4); " OPEN TELEPHONE"
210 PRINT CHR$(4); "WRITE TELEPHONE"
220 FOR J=1 TO N
230 PRINT A$(J)","B$(J)","C$(J)"," D$(J)","E$(J)","F$(J)
240 NEXT J
310 PRINT CHR$(4); "CLOSE TELEPHONE"
320 END
```

The above program may be used to set up a directory consisting of any number of entries. You should set up such a computerized telephone directory of your own. It is very instructive. Moreover, when coupled with the search program given in Example 2 below, it will allow you to look up addresses and phone numbers using your computer.

TEST YOUR UNDERSTANDING 2

Use the above program to enter the following address into the file.

John Jones
1 South Main St., Apt. 308
Phila. Pa. 19107
527-1211

The **WRITE** instruction begins to write the file from the beginning. If you wish to add to an existing file, you should also use the **APPEND** instruction:

```
10 PRINT CHR$(4); "APPEND TELEPHONE"
20 PRINT CHR$(4); "WRITE TELEPHONE"
```

Such a sequence of instructions will cause any subsequent **PRINT** statements to print their data at the end of the file. The APPEND statement substitutes for the OPEN statement.

Note that the APPEND statement assumes that there is an existing file by that name. If there is not, you will receive an error message. You can always protect yourself against this contingency by first opening the file and closing it. This will create an empty file if none exists. The APPEND statement will then put the desired data in the first record of that blank file.

Example 2. Write a program which adds entries to the file TELEPHONE. The additions should be typed via INPUT statements. The program may assume that the file is on the diskette in drive 1.

Solution. Here is the program.

```
110 PRINT "TYPE ENTRY:NAME,STREET ADDRESS,CITY, STATE,"
120 PRINT "ZIP CODE, TELEPHONE NO."
130 INPUT A$,B$,C$,D$,E$,F$
140 PRINT CHR$(4); "APPEND TELEPHONE"
150 PRINT CHR$(4); "WRITE TELEPHONE"
200 PRINT A$","B$","C$","D$","E$","F$
210 PRINT "CLOSE TELEPHONE"
250 INPUT "ANOTHER ENTRY (Y/N)": Z$
260 IF Z$="N" THEN END
300 HOME
310 GOTO 110
```

TEST YOUR UNDERSTANDING 3 (answers on page 174)

Add to the telephone file started in TEST YOUR UNDERSTANDING 2 the following entry:

Mary Bell
2510 9th St.
Phila. Pa. 19138
937-4896

WARNING: If you wish to replace a file which already exists with one of the same name, you should erase the first file using an instruction of the form

```
20 PRINT CHR$(4); "DELETE TELEPHONE"
```

Then you may safely write the new version using the instructions shown above. If you fail to delete the old version you run the risk of mixing the old and new versions. In short, you may create a ghastly mess!

Let's now discuss the procedure for reading data files from a diskette. As is the case with writing files, it is necessary to first open the file. Consider the

telephone file in Example 1. To open it for input, we could use the instruction

```
300 PRINT CHR$(4); "OPEN TELEPHONE"
310 PRINT CHR$(4); "READ TELEPHONE"
```

Once the file is open, it may be read via the instruction

```
400 INPUT A$,B$,C$,D$,E$,F$
```

This instruction will read one of the telephone-address entries from the file.

In order to read a file, it is necessary to know the precise format of the data in the file. For example, the form of the above **INPUT** statement was dictated by the fact that each telephone-address entry was entered into the file as six consecutive string constants separated by commas. The input statement works like any other input statement: faced with a list of items separated by commas, it assigns values to the indicated variables in the order in which the data items are presented. The commas in the data file are essential: in order for an INPUT statement to assign values to several variables at once, the values must be separated by commas!

Example 3. Write a program which searches for a particular entry of the telephone directory file created in Example 1.

Solution. We will **INPUT** the name corresponding to the desired entry. The program will then read the file entries until a match of names occurs. Here is the program.

```
5 INPUT "NUMBER IN DIRECTORY? ";N
10 INPUT "NAME TO SEARCH FOR"; Z$
15 PRINT CHR$(4); "OPEN TELEPHONE"
16 PRINT CHR$(4); "READ TELEPHONE"
20 FOR I=1 TO N
25 INPUT A$,B$,C$,D$,E$,F$
30 IF A$=Z$ THEN 100
40 NEXT I
50 GOTO 200
100 HOME
110 PRINT A$
120 PRINT B$
130 PRINT C$,D$,E$
140 PRINT F$
150 PRINT CHR$(4);"CLOSE TELEPHONE": END
200 HOME
300 PRINT CHR$(4);"CLOSE TELEPHONE"
400 PRINT "THE NAME IS NOT ON FILE"
1000 END
```

TEST YOUR UNDERSTANDING 4

Use the above program to locate Mary Bell's number in the telephone file created in TEST YOUR UNDERSTANDINGS 2 and 3.

In the above programs we needed to know the number of entries in the directory in order to avoid an end of data error. In Section 8.3 we will see a more sophisticated method of handling this problem.

In the next example we present a program useful for parents who wish to teach organizational skills to their children. Most children love to play with the computer. Here is a program which acts as an assignment book and monitors progress on homework. This program was designed for Jonathan Goldstein, a 10-year-old computer enthusiast.

Example 4. Write a program which sets up a data file for homework assignments. The child should enter the assignments, by subject, when returning from school. As assignments are completed, the child may check them off. The program should tell the child whether his homework is complete.

Solution. Our program will first ask if the assignment has been recorded before. If so, it will be in a file. If not, the program will prompt the child to type in the assignments by subject, with prompts like

```
What is your math assignment?
```

The only rule is that assignments cannot have commas in their statements. The child types in the assignment followed by RETURN and the computer responds with the next subject. If there is no assignment, type RETURN. (You may customize the program by entering your own subjects.)

After all assignments are entered, the computer asks the child if he wishes to have the assignments displayed. If the answer is yes, the computer then produces a list of all subjects with their corresponding assignments.

```
SUBJECT     ASSIGNMENT

MATH        P45 1-20
READING     CHAP 3
SPELLING    P80 SENTENCES
.
.
.
```

The computer now gives the child a chance to check off a completed assignment. The computer doesn't allow for forgetfulness. It asks for the subject and then displays the assignment and asks if, in fact, that assignment has been completed. If so, the next time the list of assignments is displayed an X will appear beside completed assignments. Finally, the computer scans the list of assignments and decides whether all are complete. If so, it prints "HOMEWORK DONE"; if not, it prints "HOMEWORK NOT DONE". Here is the program.

```
20 DIM B$(20),C$(20),D$(20)
30 HOME
40 PRINT "HAVE YOU ENTERED THIS ASSIGNMENT BEFORE?"
50 INPUT A$
60 IF A$="Y" THEN 70
65 GOTO 140
70 PRINT "SUBJECT";TAB(20) "ASSIGNMENT"
```

```
80 PRINT CHR$(4); "OPEN SCHED"
85 PRINT CHR$(4); "READ SCHED"
90 FOR J=1 TO 7
100 INPUT B$(J),C$(J),D$(J)
110 PRINT B$(J);TAB(15) C$(J);TAB(25) D$(J)
120 NEXT J
130 PRINT CHR$(4); "CLOSE SCHED"
135 GOTO 370
140 HOME
150 DATA "MATH", "SPELLING", "LANGUAGE","SOCIAL STUDIES"
160 DATA "READING", "SCIENCE", "CURRENT EVENTS"
170 FOR J=1 TO 7
180 READ B$(J)
190 PRINT "DO YOU HAVE ANY ";B$(J);" HOMEWORK TONIGHT?"
200 INPUT A$
210 IF A$="Y" THEN 220
215 GOTO 250
220 PRINT "WHAT IS THE ";B$(J);" ASSIGNMENT?"
230 INPUT C$(J)
250 NEXT J
260 PRINT "DO YOU WISH TO SEE YOUR ASSIGNMENTS?"
270 INPUT A$
280 IF A$="Y" THEN 300
290 GOTO 370
300 HOME
310 PRINT "SUBJECT";TAB(15) "ASSIGNMENT"
320 PRINT
330 FOR J=1 TO 7
340 PRINT B$(J); TAB(15) C$(J)
350 NEXT J
370 PRINT "DO YOU WANT TO CHECK OFF AN ASSIGNMENT?"
380 INPUT A$
400 IF A$="Y" THEN 410
405 GOTO 520
410 INPUT "SUBJECT";B$
420 FOR J=1 TO 6
440 PRINT "IS THIS ASSIGNMENT DONE?"
450 PRINT C$(J)
460 INPUT A$
470 IF A$="Y" THEN D$(J)="X"
475 IF B$=B$(J) THEN PRINT B$(J) : NEXT J
480 HOME
490 FOR J=1 TO 7
500 PRINT B$(J);TAB(15) C$(J);TAB(25) D$(J)
510 NEXT J
520 FOR J=1 TO 7
530 IF D$(J) <> "X" AND C$(J) <> "" THEN E$="W"
540 NEXT J
550 IF E$="W" THEN PRINT "HOMEWORK IS NOT DONE"
555 IF E$"<>"W" THEN PRINT "HOMEWORK DONE"
560 INPUT "DO YOU WISH TO CHECK OFF ANOTHER ASSIGNMENT";A$
570 IF A$="Y" THEN GOTO 380
```

```
575 PRINT CHR$(4); "DELETE SCHED"
580 PRINT CHR$(4); "OPEN SCHED"
581 PRINT CHR$(4); "WRITE SCHED"
590 FOR J=1 TO 7
600 PRINT B$(J)","C$(J)","D$(J)
610 NEXT J
620 PRINT CHR$(4); "CLOSE SCHED"
700 END
```

The files we have been discussing are called *sequential files*. They must be read in the exact order in which they are written.

DOS also allows you to have *random access* files. These files let you read a given piece of data without reading all the data written ahead of it. We will discuss random access files in Section 7.5 after looking at a couple of examples of the use of sequential files.

Exercises (answers on page 318)

1. Write a program creating a diskette data file containing the numbers 5.7, −11.4, 123, 485, and 49.
2. Write a program which reads the data file created in Exercise 1 and displays the data items on the screen.
3. Write a program which adds to the data file of Exercise 1 the data items 5,78, 4.79, and −1.27.
4. Write a program which reads the expanded file of Exercise 3 and displays all the data items on the screen.
5. Write a program which records the contents of checkbook stubs in a data file. The data items of the file should be as follows:

   check #, date, payee, amount, explanation

   Use this program to create a data file corresponding to your previous month's checks.
6. Write a program which reads the data file of Exercise 5 and totals the amounts of all the checks listed in the file.
7. Write a program which keeps track of inventory in a retail store. The inventory should be described by a data file whose entries contain the following information:

   item, current price, units in stock

   The program should allow for three different operations: display the data file entry corresponding to a given item, record the receipt of a shipment of a given item, and record the sale of a certain number of units of a given item.
8. Write a program that creates a recipe file to contain your favorite recipes.
9. (For teachers) Write a program that maintains a student file containing your class roll, attendance, and grades.

10. Write a program maintaining a file of your credit card numbers and the party to notify in case of loss or theft.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1 and 3

1: The data file name is TRIAL. It will contain the following (the downward arrow represents a carriage return):

```
THIS IS A TEST ↓10 PRINT "HELLO"↓20 PRINT "GOODBYE"↓
```

This results from statement 50 being executed and thus printing the sentence "THIS IS A TEST" onto the file, followed by a listing of lines 10 and 20 being written into the file.

3: To add to an existing file, you must change line 200 to an APPEND statement and change the prompt message in line 10 to request the number to be added.

7.3 Application 7-1: Tracking Monthly Bills

In this section we will develop a rather lengthy application program that will use sequential files as well as many of the other techniques we have discussed. These include menu selection and subroutines.

The program will maintain a permanent file of monthly expenses for a variety of accounts which you choose. It will allow you to enter transactions against any account during the current month and to keep a list of all current transactions. At the end of the month, it will display the list of transactions for each account and then delete the transaction files to prepare for the start of the next month. In case you make an error in an amount, you can go back and change the monthly totals for any account. The transaction list for the current month remains unaffected.

We will need to maintain several files. One of these will be the permanent file containing the monthly expenses. The first two pieces of data in the file will be the number of accounts and the current month number. After that, each account will have associated with it a name, the total number of transactions in the current month, and an expense total for each month. This will be 0 if we have not yet reached that month. We will also have a transaction file for each account against which there have been any transactions during the current month. We will call the permanent expense file PERMEX and the transaction files T1 for account number 1, T2 for account number 2, and so on. We will keep the month names in an array which we will READ in.

The major blocks in the solution procedure are:

```
initialization
main menu selection
adding, deleting, altering account names
adding a transaction
changing a monthly account total
```

```
displaying accounts and totals
starting a new month
ending the session
```

Let's take each of these in turn and flesh them out.

```
initialization

     read data from the PERMEX file into memory
     read month names into an array
```

We will define the following variables:

```
N          number of accounts

M          current month number

T$(30)     account names (if you need more accounts than
           this you can simply change the dimension)

T(30)      number of transactions for each account in the
           current month

M1(30,12)  account total for each account in each of 12
           months

M$(12)     name of each month

MT(12)     total of all expenses for each month

main menu selection
     display the options:
          adding, deleting, altering account names
          adding a transaction
          changing a monthly account total
          displaying accounts and totals
          starting a new month
          ending the session

     input a choice

adding, deleting, altering account names

     display change menu:
          add an account
          delete an account
          change an account name
          return to the main menu

     input selection
     add an account
          increase number of accounts by one
          input amounts for each available month

     delete an account
        because transaction files are keyed to account
             numbers, accounts can only be deleted
             immediately after the start of a new month.
             Therefore we must check to see if the start
             of a new month just occurred
```

```
      if so, get number of account to delete
      verify deletion request
      if verified
           decrease number of accounts by one
           adjust T$ and M1 arrays

      change an account name
           get number of account to change
           ask for new name

      return to the main menu

 adding a transaction
      get the account number for the transaction
      get the amount of the transaction
      increase transaction account by one
      get date of month
      get description of expense
      write date, description, and amount on transaction
      file
      return to main menu

 changing a monthly account total

      get account number, month, and present amount
      get new amount
      continue changes or return to main menu

 displaying accounts and totals
      calculate monthly totals
      two consecutive months can be displayed on a 40
           column screen, thus we must get the first and
           last months to be displayed.
      print account name, monthly total for selected month(s)
      print account totals
      return to main menu

 starting a new month
      calculate monthly totals
      print total expenses for each active account for month
      loop for each account
           print transaction data for each active account
      endloop
      verify start of new month
      if verified
           delete all transaction files
           increase month number by one
           if new month number exceeds 12, set it back to 1
           set new month flag to allow for account deletion

 ending the session
      write all information to PERMEX
      end
```

There is one activity which appears in two of these blocks. That is the routine to calculate monthly totals. We will make this a subroutine also.

```
calculate monthly totals

     loop for each month
          add totals for all accounts
     endloop
```

Now we can construct our BASIC program. We will arrange our subprograms in order of expected frequency. Thus we make the following assignments:

initialization	10000-11000
main menu selection	300-499
adding, deleting, altering account names	4000-4999
adding a transaction	500-899
changing a monthly account total	2000-2999
displaying accounts and totals	1000-1199
starting a new month	5000-5999
ending the session	9000-9999
calculate monthly totals	900-999

The program starts with GOSUBs to the initialization subroutine and to the calculate monthly totals subroutine. Follow each subroutine to see that you understand the logic. Then try running the program. Note that because the system always keeps data on all twelve months, it can also be used to prepare a budget by entering budgeted amounts.

```
10  GOSUB 10000: GOSUB 900
300  REM   MENU
310  HOME : PRINT "CURRENT MONTH IS ";M$(M)
320  PRINT : PRINT : PRINT
330  PRINT "TYPE IN  T TO ADD A TRANSACTION"
340  PRINT "         D TO DISPLAY ACCOUNTS & TOTALS"
350  PRINT "         C TO CHANGE ACCOUNT TOTAL"
360  PRINT "         A TO ALTER ACCOUNT NAMES"
370  PRINT "         Q TO END THIS SESSION"
380  PRINT "   OR    M TO START A NEW MONTH"
390  GET A$: HOME
395  IF A$="T" THEN GOSUB 500: GOTO 300
400  IF A$="D" THEN  GOSUB 900: GOSUB 1000: GOTO 300
410  IF A$="C" THEN  GOSUB 2000: GOTO 300
420  IF A$="A" THEN  GOSUB 4000: GOTO 300
430  IF A$="Q" THEN  GOTO 8000
440  IF A$="M" THEN  GOSUB 900: GOSUB 9000
450  GOTO 300

500  REM  ADD A TRANSACTION
510  PRINT "ENTER THE ACCOUNT NUMBER OR 0 TO ABORT ":
       INPUT II
520  IF II=0 THEN  RETURN
530   IF II > N THEN PRINT "THIS IS NOT A VALID ACCOUNT
          NUMBER": GOTO 6010
```

```
600  PRINT "FOR ";T$(II);"  TYPE IN"
610  PRINT "AMOUNT": INPUT AA
620  M1(II,M)=M1(II,M) + AA
630  PRINT : PRINT : PRINT
640  PRINT : PRINT : PRINT

700  REM  APPEND TO MONTH ACCOUNT
710  PRINT : PRINT : PRINT "FOR "T$(II);" AN AMOUNT OF
          $";AA
720  PRINT "WAS ENTERED IN MONTH ";M
730 T(II)=T(II) + 1
740  PRINT :  PRINT :  PRINT "WHAT IS THE DATE OF THE
          MONTH?  ": INPUT D
750  PRINT : PRINT : PRINT "DESCRIPTION ?": INPUT E$
760  PRINT D$;"OPEN T";II
770  PRINT D$;"APPEND T";II
780  PRINT D$;"WRITE T";II
790  PRINT D: PRINT E$: PRINT AA
800  PRINT D$;"CLOSE T";II
810  RETURN

900  REM  ADD TO FIND NET
910  FOR I=1 TO 12:S=0: FOR K=1 TO N
920 S=S + M1(K,I): NEXT K:MT(I)=S
930  NEXT I: RETURN
1000  REM  PRINT INFO
1010  PRINT "A MAXIMUM OF 2 MONTHS CAN BE DISPLAYED"
1011  PRINT
1020  PRINT "ON THE SCREEN.  PLEASE TYPE IN THE"
1021  PRINT
1030  PRINT "NUMBERS OF THE FIRST AND LAST MONTHS"
1031  PRINT
1040  INPUT " WHICH YOU WISH TO SEE";L,LL
1050  HOME
1060   PRINT : PRINT : PRINT :C=10: PRINT "   MONTH";:
          IF LL - L > 2 THEN LL=L + 1
1070 Z=0:  FOR I=L TO LL:  PRINT TAB( 21 + C * Z);I;:
          Z=Z + 1: NEXT I: PRINT
1080  PRINT : PRINT
1090  FOR K=1 TO N:Z=0
1100  PRINT K;"  ";T$(K);
1110  FOR I=L TO LL
1130  PRINT  TAB( 21 + C * Z);M1(K,I);:Z=Z + 1: NEXT I
1140  PRINT : NEXT K
1150  PRINT : PRINT "TOTAL";:Z=0
1160  FOR I=L TO LL
1170  PRINT  TAB( 21 + C * Z);MT(I);:Z=Z + 1: NEXT I:
          PRINT
1180  PRINT : PRINT : PRINT "HIT RETURN TO CONTINUE": GET
          A$: HOME
1200  RETURN

2000  REM  CHANGE AN ACCOUNT TOTAL
```

```
2010  PRINT : PRINT "YOU WILL BE ASKED FOR THE ACCOUNT #"
2020  PRINT : PRINT "THE MONTH #, AND THE NEW AMOUNT."
2030  PRINT : PRINT "HIT R TO RETURN TO MAIN MENU"
2040  PRINT : PRINT "    C TO CONTINUE CHANGES": GET B$
2050  IF B$="R" THEN  RETURN
2060  HOME : INPUT "WHAT IS NUMBER OF ACCOUNT TO BE
        CHANGED? ";A
2070  PRINT : PRINT "THIS IS ";T$(A)
2080  PRINT : INPUT "CONTINUE? (Y/N)";B$
2090  IF B$="N" THEN 2030
2100  PRINT : INPUT "CHANGE FOR WHAT MONTH NUMBER? ";B
2110  PRINT : PRINT "TOTAL IS NOW ";M1(A,B)
2120  INPUT "WHAT IS NEW AMOUNT? ";M1(A,B)
2125 F(C)=1
2130  PRINT "ACCOUNT ";"# ";A;" ";T$(A);" FOR ""MONTH ";B;
        "  IS NOW ";M1(A,B)
2140  GOTO 2030


4000  HOME: REM   TO ALTER,ADD OR DELETE ACCOUNT
4010  VTAB (10): PRINT "HIT A - TO ADD AN ACCOUNT"
4012  PRINT "    D - TO DELETE AN ACCOUNT"
4014  PRINT "    C - TO CHANGE AN ACCOUNT NAME"
4016  PRINT "OR  R - TO RETURN TO MAIN MENU": GET Z$
4020  IF Z$="C" THEN GOTO 4300
4030  IF Z$="D" THEN GOTO 4100
4040  IF Z$="R" THEN  RETURN
4042  IF Z$="A" THEN GOTO 4050
4045 GOTO 4000
4050 N=N + 1: PRINT : PRINT "ACCOUNT NAME?  ": INPUT T$(N)
4060  PRINT "THIS IS ACCOUNT # ";N
4070   PRINT "MONTH # AND AMOUNT?  (0,0 WHEN FINISHED)  ":
        INPUT I,AA
4080  IF I=0 GOTO 4000
4090 M1(N,I)=AA: GOTO 4070
4100 IF FLAG=1 THEN GOTO 4150
4110 HOME: PRINT "ACCOUNTS CAN ONLY BE DELETED"
4120 PRINT "AFTER STARTING A NEW MONTH"
4130 PRINT "PLEASE MAKE ANOTHER SELECTION": GOTO 4010
4150  INPUT "ACCOUNT NUMBER TO BE DELETED?  ";A
4155  PRINT : PRINT : PRINT T$(A);" WILL BE DELETED."
4157  PRINT : PRINT
4160  PRINT:  PRINT:  PRINT "TYPE C TO CONTINUE OR Q TO
        CANCEL": GET Z$
4165  IF Z$="Q" GOTO 4000
4170  IF Z$ <> "C" GOTO 4155
4180  N=N - 1: FOR K=A TO N:T$(K)=T$(K + 1):C(K)=C(K + 1):
       FOR I=1 TO 12:M1(K,I)=M1(K + 1,I): NEXT I: NEXT K
4190  GOTO 4000
4300  INPUT "WHAT NUMBER ACCOUNT TO CHANGE? (0 TO ABORT)";A
4310  IF A > N THEN PRINT "NOT A VALID ACCOUNT NUMBER":
        GOTO 4300
4315 IF A=0 THEN GOTO 4000
```

```
4320  PRINT : PRINT "ACCOUNT IS NOW ";T$(A)
4330  PRINT : INPUT "WHAT IS THE NEW NAME? ";T$(A)
4340  GOTO 4000

8000  REM QUIT ROUTINE
8010  PRINT "TYPE Q TO QUIT, R TO RETURN TO MENU": GET A$
8020  IF A$="R" GOTO 330
8030  IF A$ <> "Q" GOTO 8010
8100  REM    WRITE TO DISK
8120  PRINT D$;"DELETE PERMEX"
8130  PRINT D$;"OPEN PERMEX"
8140  PRINT D$;"WRITE PERMEX"
8150  PRINT N: PRINT M
8160  FOR K=1 TO N
8170  PRINT T$(K): PRINT T(K): FOR I=1 TO 12
8180  PRINT M1(K,I): NEXT I: NEXT K
8190  PRINT D$;"CLOSE PERMEX"
8200  PRINT "THE NEW ACCOUNT BALANCES HAVE BEEN SAVED.
          GOOD-BYE"
8500  END

9000  REM  CLOSING OUT THE MONTH
9010  PRINT "ACCOUNT TOTALS FOR ";M$(M): PRINT : PRINT
9120  FOR K=1 TO N
9130  IF M1(K,M)=0 THEN  GOTO 9150
9140  PRINT T$(K); TAB( 25);M1(K,M)
9150  NEXT K: PRINT : PRINT
9170  PRINT  SPC( 10);"TOTAL"; TAB( 24);"$";MT(M)
9180  PRINT : PRINT : PRINT : PRINT : PRINT
9190  PRINT "HIT RETURN TO CONTINUE": GET A$: HOME
9240  FOR K=1 TO N
9250  IF T(K)=0 THEN  GOTO 5410
9260  PRINT T$(K);" FOR ";M$(M)
9270  PRINT : PRINT "DATE    DESCRIPTION"; TAB( 31);"AMOUNT"
9280  PRINT D$;"OPEN T";K
9290  PRINT D$;"READ T";K
9300  FOR J=1 TO T(K)
9310  INPUT D,E$,A
9340  PRINT D; TAB( 5);E$; TAB( 30);A
9350  NEXT J
9400  PRINT D$;"CLOSE T";K: PRINT : PRINT
9405  PRINT "HIT RETURN TO CONTINUE": GET A$: HOME
9410  NEXT K: PRINT : PRINT : PRINT
9500  PRINT "SHOULD A NEW MONTH BE STARTED? "
9510  PRINT :  PRINT "CAUTION, IF YES,":
          PRINT "ALL TRANSACTION FILES WILL BE DESTROYED"
9520  PRINT :  PRINT "NEW MONTH OR RETURN TO MAIN  MENU?
          (N/R)": GET A$
9530  IF A$="R" THEN  RETURN
9540  IF A$ <> "N" THEN 9500
9550  FLAG=1
9600  FOR K=1 TO N
9605  IF T(K)=0 THEN 9620
```

```
9610  PRINT D$;"DELETE T";K:T(K)=0
9620  NEXT K
9630 M=M + 1: IF M > 12 THEN M=1
9640  RETURN
10000 D$=CHR$ (4): DIM T$(30),T(30),M1(30,12),
          M$(12),MT(12)
10010  PRINT D$;"OPEN PERMEX"
10020  PRINT D$;"READ PERMEX"
10030  INPUT N,M: IF N=0 THEN 10070
10050  FOR K=1 TO N: INPUT T$(K)
10055  INPUT T(K)
10060  FOR I=1 TO 12: INPUT M1(K,I): NEXT I: NEXT K
10070  PRINT D$;"CLOSE PERMEX"
10100  FOR K=1 TO 12: READ M$(K): NEXT K
10110  DATA JANUARY,FEBRUARY,MARCH,APRIL,MAY,JUNE,
          JULY,AUGUST,SEPTEMBER,OCTOBER,NOVEMBER,DECEMBER
10150  RETURN
```

Initial File Set-Up

Before running the program for the first time, it is necessary to create the PERMEX file. We will do that by writing an initial value of 0 for the N (number of accounts) and a value for M (the month number) which is input during the initial set up. Valid accounts can then be added the first time the program is run. Here is the set-up program.

```
5 INPUT "START WITH WHAT MONTH? ";M
10 PRINT CHR$(4);"OPEN PERMEX"
20 PRINT CHR$(4);"WRITE PERMEX"
30 PRINT 0:  PRINT M
40 PRINT CHR$(4);"CLOSE PERMEX"
```

Enhancements and Alterations

Remember that the program is designed for a 40-column screen. If you wish to use it on an 80-column screen, you will have to change the HOME and TAB statements. Similarly, if you have a printer attached, you might wish to add another routine to display accounts and totals on the printer. In this case you would also want to alter the start for a new month routine so that the transaction listing is printed out. This would allow you to keep a permanent hard copy record.

If you wish to keep a permanent file of your transactions on disk, you should alter this program by adding line number 9607 to read

```
9607 PRINT D$;"OPEN T";K
```

This will insure that a file is present when you try to delete. With this line in place, you can use the RENAME command (see Section 2.6) to save the

transaction files under another name. RENAME the transaction files just before running the program to start a new month.

Alternatively, this can be built right into the program. For example, assume that you wanted to save the transaction file for month 3 and account 2 under the name T2-3. Then you could change line 9610 to read

```
9610 PRINT D$;"RENAME T";K;",T";K;"-";M
```

This will rename instead of deleting.

7.4 Application 7-2: Attaching Subroutines to Other Programs

In Chapter Six we introduced subroutines as a good way to subdivide your program into manageable modules. In addition, there are frequently small sections of code which you may wish to use over and over again, perhaps with some minor modifications. It is already very helpful to be able to use the same code again, but it would be even better if we didn't have to type it in again either. Using sequential data files, or text files as they are sometimes called, we can accomplish this.

You will remember that when a file is opened for writing, anything which would normally appear on the screen is written into a file. This means that if we put a **LIST** command into the BASIC program, for example,

```
LIST 90,100
```

all the program lines between 90 and 100 would be written into our file. This means that we can make a text file that consists of program statements.

The value of this is not apparent until we explain another DOS command, namely the **EXEC** command. This command, followed by the name of a text file, causes the commands in the text file to act as if they had been typed in at the keyboard. This means that if the text file contains BASIC statements, they will be loaded into memory exactly as if you had typed them in.

If you create a text file which contains a BASIC subroutine, you can add that subroutine to your program simply by EXECing that text file.

For example, let's consider the menu routine which we developed in Section 5.5. This routine is very useful, and we will probably want to use it many times. Let's give the routine line numbers starting at 9000 so that it is out of the way of the rest of the program and add a remark reminding us what has to be changed to customize it for individual programs.

```
9000 REM CHANGE LINES 9040-9070,9090,9100
9010 HOME
9020 PRINT "CHOOSE ONE OF THE FOLLOWING:"
9030 PRINT
9040 PRINT TAB(5);"1 -- FIRST ROUTINE "
9050 PRINT TAB(5);"2 -- SECOND ROUTINE "
9060 PRINT TAB(5);"3 -- THIRD ROUTINE "
9070 PRINT TAB(5);"4 -- FOURTH ROUTINE "
```

```
9080 PRINT: PRINT "TYPE IN ONE NUMBER ";: GET A: PRINT A
9090 ON A GOTO 1000, 3000, 450, 3400
9100 FLASH: VTAB(15): PRINT TAB(5);"ONLY VALUES FROM 1 TO 4
     ALLOWED": NORMAL
9110 FOR K=1 TO 2000: NEXT K
9120 VTAB(15): PRINT SPC(40)
9130 VTAB(8): HTAB(20): PRINT " ": VTAB(8): HTAB(20): GET
     A: PRINT A: GOTO 9090
```

Now let's put this code into a text file called **MENU**. We can create such a text file by creating a file, opening it for writing, and then listing lines 9000 through 9130. The program to do this is

```
1 REM TYPE LINE NUMBERS IN LINE 5
2  INPUT "NAME OF FILE? ";A$
3  D$=CHR$(4): PRINT D$;"OPEN";A$
4  PRINT D$;"WRITE";A$
5 LIST 9000,9130
6 PRINT D$;"CLOSE";A$
7 END
```

If we run this program with our menu routine in lines 9000 through 9130, calling the file MENU, the program will create a text file containing our menu routine. To attach this routine to any program all we need to do once we have the program in memory is to type in

```
EXEC MENU
```

The menu routine will then be attached to our program at line 9000 without destroying the current program in memory.

We can take any frequently used routine or subroutine and put it into a text file for later attachment to new programs. In fact, we can take the program in lines 1 through 7 above and put it into a text file also, perhaps one called MAKE-TEXT. Then as we write our new routine and find that we wish to save it as a text file for future use, all we need to do is EXEC MAKE-TEXT and RUN.

Here is the complete program to create a MENU text file.

```
1 REM TYPE LINE NUMBERS IN LINE 5
2  INPUT "NAME OF FILE? ";A$
3  D$=CHR$(4): PRINT D$;"OPEN";A$
4  PRINT D$;"WRITE";A$
5 LIST 9000,9130
6 PRINT D$;"CLOSE";A$
7 END
9000 REM CHANGE LINES 9040-9070,9090,9100
9010 HOME
9020 PRINT "CHOOSE ONE OF THE FOLLOWING:"
9030 PRINT
9040 PRINT TAB(5);"1 -- FIRST ROUTINE "
9050 PRINT TAB(5);"2 -- SECOND ROUTINE "
9060 PRINT TAB(5);"3 -- THIRD ROUTINE "
9070 PRINT TAB(5);"4 -- FOURTH ROUTINE "
```

```
9080 PRINT: PRINT "TYPE IN ONE NUMBER ";: GET A: PRINT A
9090 ON A GOTO 1000, 3000, 450, 3400
9100 FLASH: VTAB(15): PRINT TAB(5);"ONLY VALUES FROM 1 TO 4
     ALLOWED": NORMAL
9110 FOR K=1 TO 2000: NEXT K
9120 VTAB(15): PRINT SPC(40)
9130 VTAB(8): HTAB(20): PRINT " ": VTAB(8): HTAB(20): GET
     A: PRINT A: GOTO 9090
```

Type this program into memory. Then RUN it giving the file the name of MENU. Save the program so that you can later create a MAKE-TEXT text file.

When you have done this, type in NEW. Then LIST to see that there is nothing in memory. When you have done this type in

```
EXEC MENU
```

Now LIST it (remember that on the Apple IIe you can LIST simply by hitting the ESC key and then the L key). You will see that the menu subroutine is in memory now. If there had been a program in memory at that time these lines would have been added to that program.

CAUTION: Remember that the EXEC command will load the lines in exactly as if you had typed them from the keyboard. This means that if you had line numbers in your program from 9000 through 9130, they would have been overwritten. This is one good reason for planning your program, including the line numbers you are going to use for each section, in advance.

7.5 Using Random Access Files

In the previous sections we talked about and used sequential files. The data always had to be accessed one after the other. Many applications require that we enter the file at random points to look at selected entries. For these purposes we need a random access file.

In discussing sequential files we were only concerned with the individual data items because this was the only way that we could access the data. The sequential file stored the data items one after the other with each data item being separated by a delimiter, either a comma or a carriage return. Each piece of data occupied as much room as it needed and no more.

In a random access file we wish to be able to go directly to the start of any set of data. That means that we need to know exactly where to find the required data. To accomplish this we organize each set of data items into what is called a "record."

For example, if we were storing inventory information, the product item number, description, price, and number in stock might form one data record containing four items. We would have a record like this for each item in inventory. The set of four items is called a **record**; each separate item is referred to as a **data item**, **data element**, or sometimes a **field**; the set of all records is a **file**.

In order to find the start of a record, the computer needs to know the record length. Record length is defined by the number of characters it contains. This count should include room for a delimiter after each data item. For example, if in the above example of inventory information the item number required 4 characters, description was 30 characters, price was 6 and number in stock used 4 characters, we would want to set the record length at 4 + 30 + 6 + 4=44 characters plus 4 additional characters for a delimiter after each data item. Thus, the record length would be 48 characters. We define the record length when we open the file. The BASIC instruction for opening a random file with record length of 48 is

```
100 PRINT CHR$(4); "OPEN INVEN,L48"
```

As in using a sequential file, the name of the file is used in the OPEN statement; however, in the case of a random access file, we also add the letter L followed by the record length. Remember that CHR$(4) stands for a CONTROL-D sequence which tells the computer this is a DOS command.

To read or write to a random file, we need to tell the computer which record to read. We do this by adding a record number to the READ or WRITE file statement. For example

```
110 PRINT CHR$(4); "READ INVEN,R12"
```

instructs the computer to prepare record number 12 for reading. Again, no reading of the file actually occurs until we provide an INPUT statement and a variable list. The records in a file are numbered beginning with record 0.

Writing a file is done in a similar fashion. We prepare the file for writing by using a command like

```
150 PRINT CHR$(4); "WRITE INVEN,R12"
```

No writing to the file is actually done until we provide PRINT statements or otherwise include a statement which would normally display something on the screen.

In addition to reading or writing a specific record, we can also read or write to any specific byte (character) in the record (you will remember that a byte is the storage requirement for storing one character). Therefore, if we wanted to be sure that the price always began at character 36 (bytes or characters are numbered starting with byte 0 at the left) we could write this information as

```
100 PRINT CHR$(4); "WRITE INVEN, R12, B36"
```

Using the byte option in the read or write statement makes the program somewhat more difficult to understand and more tedious to write because we must know exactly where we are writing each data item. However, the use of the byte option can simplify the execution of many programs. We will use this option in the application program given in Section 7.7.

The CLOSE statement for random files is identical to the CLOSE statement for sequential files:

```
160 PRINT CHR$(4); "CLOSE INVEN"
```

Example 1. Inventory management: write a program which will keep a file of all the parts stocked by a company, allowing the user to look at the records on any part at random, and which will allow the file to be updated as parts are sold or restocked.

Solution. Let's set up a file as described above with I as part number, DS$ as description, P$ as price, and N$ as number in stock.

```
initialize: get the desired part number

read: read data about that part from the file

select: ask whether any changes are to be made?
     if no, goto initialize
     if yes, get new data
             write new data on file
             goto initialize
```

We assume that the part numbers can be used as a key to the file, or part number 34 is stored at record 34, and so on.

```
10 HOME: D$=CHR$(4)
20 INPUT "WHAT PART NUMBER DO YOU WISH TO REVIEW? ";I
30 PRINT D$;"OPEN INVEN,L48"
40 PRINT D$;"READ INVEN,R";I
50 INPUT I,DS$,P$,N$
60 PRINT D$;"CLOSE INVEN"
70 HOME: VTAB(7): PRINT "FOR ITEM # ";I;":"
80 PRINT: PRINT DS$
90 PRINT: PRINT "THE CURRENT PRICE IS ";P$
100 PRINT: PRINT "NUMBER IN STOCK IS ";N$
110 VTAB(17): INPUT"DO YOU WISH TO MAKE CHANGES (Y/N)?";A$
120 IF A$="N" THEN GOTO 10
130 IF A$ <> "Y" THEN GOTO 110
140 VTAB(17):PRINT "CHANGE DESCRIPTION (D), PRICE (P), OR
     NUMBER IN STOCK (N)? "
150 INPUT "ENTER D, P, OR N ";A$
160 IF A$="D" THEN INPUT "NEW DESCRIPTION ";DS$
170 IF A$="P" THEN INPUT "NEW PRICE? ";P$
180 IF A$="N" THEN INPUT "NUMBER IN STOCK? ";N$
190 HOME: VTAB(7): PRINT "FOR ITEM # ";I;":"
200 PRINT: PRINT DS$
210 PRINT: PRINT "THE CURRENT PRICE IS ";P$
220 PRINT: PRINT "NUMBER IN STOCK IS ";N$
230 PRINT: INPUT "ANY MORE CHANGES? ";A$
240 IF A$="Y" THEN GOTO 140
250 PRINT D$;"OPEN INVEN,L48"
260 PRINT D$;"WRITE INVEN,R";I
270 INPUT I;",";DS$;",";P$;",";N$
280 PRINT D$;"CLOSE INVEN"
290 HOME: INPUT "LOOK AT MORE PARTS? ";A$
300 IF A$="Y" THEN HOME: GOTO 20
310 END
```

Exercises (answers on page 320)

1. Write a program creating a random access data file using the following DATA statement.

   DATA 5.7, -11.4, 123, 485, 49

   Put the values in the following records:

record	value
5	5.7
4	-11.4
3	123
2	485
1	49

2. Write a program which inputs a record number and then reads the contents of that record and displays the data on the screen. Use the data file created in Exercise 1.
3. Using the data file created in Exercise 1, change the contents of record 3 to 428.
4. Write a program which records the contents of checkbook stubs in a random access data file. Keep the number of the last check in record 0. Use the check number to indicate the record in which information about that check is stored. For example, if the checks are numbered starting with check number 100, then store information about that check in record 1. That means that to find the record where the information is stored, we take the check number and subtract 99. Information about check number 100 is stored in record 100 − 99 = 1; information about check number 352 is stored in record 352 − 99 = 253. The information to be stored is date, payee, amount, and explanation. It is not necessary to store the check number because we can obtain this directly from the record number.
5. Write a program to read a record from the data file developed in Exercise 4. Input the number of the check about which information you want, read the appropriate record, and display the data on the screen.

7.6 Application 7-3: A Random Access Phone Directory

In this section we will write a telephone directory program similar to the one developed in Examples 1, 2, and 3 of Section 7.2 but using a random access file. The program will be able to add names to the file, update the file, and locate the phone number for any person. To simplify the exposition a little, we will begin by writing a program for a file that contains only the name of the individual and the phone number. Later we will discuss how to modify it to include addresses.

Let's first assume we want to provide 25 characters (including delimiter) for the last name, 15 characters (including delimiter) for the first name and 13 characters for the phone number (xxx-xxx-xxxx plus a delimiter). This means we need 53 characters or bytes for the phone number information.

We also want to keep these numbers in alphabetical order. There are many ways to do this. What we would like to avoid is the necessity of rewriting the file or large sections of the file each time we need to add a new record. The technique we will use to keep the records in alphabetical order is the inclusion in each record of a pointer to the record that follows it in alphabetical order. For example, assume we have the following seven records:

record	name
1	LOIS
2	LOUISE
3	LARSON
4	LARRY
5	LOLA
6	LISA
7	LUANNE

All of these names begin with L, but they are not in alphabetical order. We could rewrite the whole list, but instead, let's just add another column to tell us where to find the next entry. The first entry on the list is LARRY, so our list will start at record number 4.

```
first record=4
record        name        pointer
  1          LOIS           5
  2          LOUISE         7
  3          LARSON         6
  4          LARRY          3
  5          LOLA           2
  6          LISA           1
  7          LUANNE         0
```

We read that record number four contains the name LARRY and a pointer to record number 3. We proceed to record number 3 and read LARSON which is the next name in alphabetical order. The pointer is to record 6 which contains the name of LISA who is next in order. The method proceeds like this until we arrive at record number 7 which has a pointer of 0. This value for our pointer indicates the end of the list since we are not storing a name at record 0. This is called a **linked list**. Schematically, our pointers look like Figure 7-1.

Now suppose we wanted to add another name, LESLIE, to the list. We could search through the list, starting from the beginning, until we found where this new name fit. In doing this, we see it belongs between LARSON and LISA. What we need to do now is to adjust the pointers so that LARSON will now point to LESLIE and LESLIE will point to LISA. The new list is:

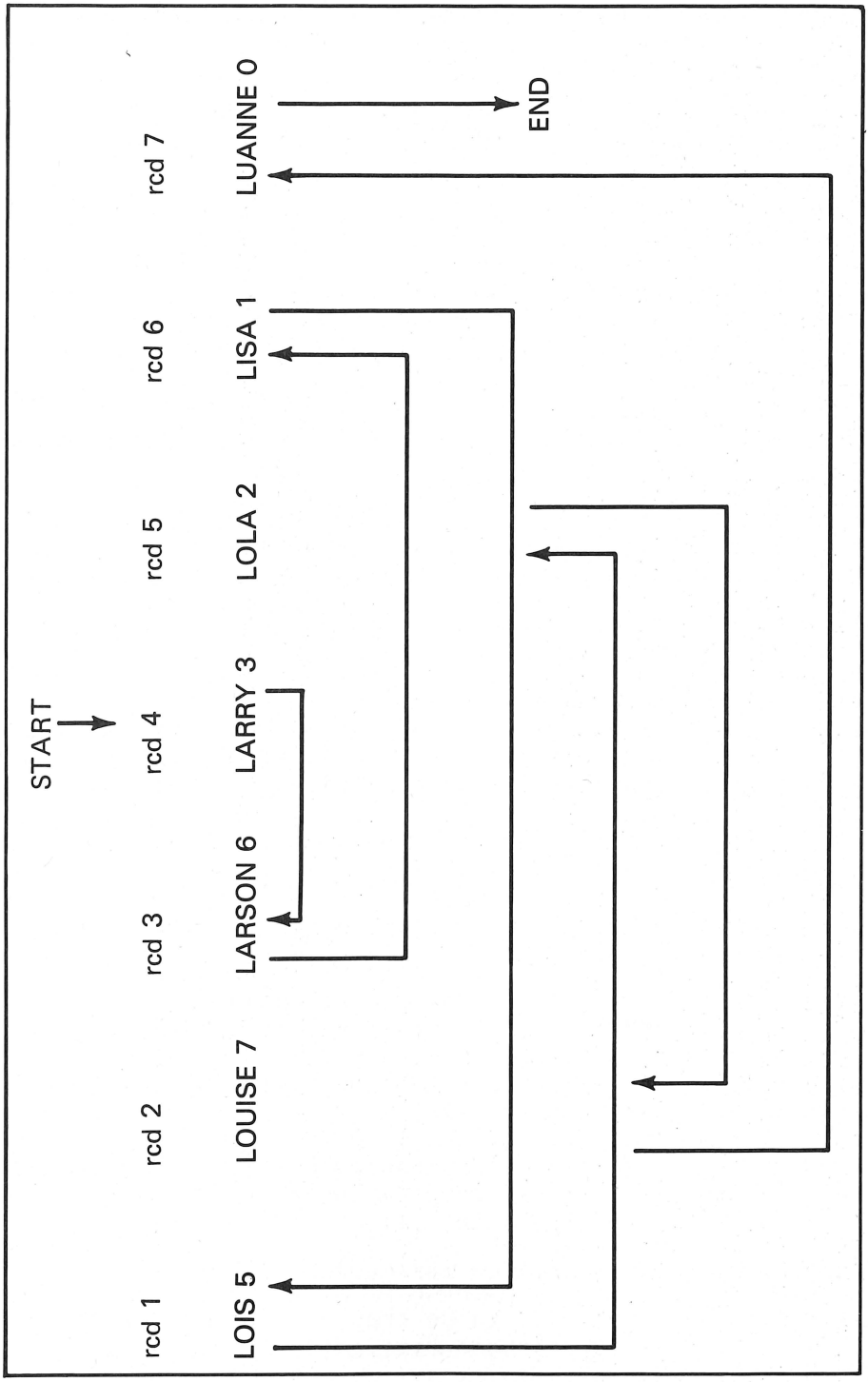

START
rcd 1
rcd 2
rcd 3
rcd 4
rcd 5
rcd 6
rcd 7
LOIS 5
LOUISE 7
LARSON 6
LARRY 3
LOLA 2
LISA 1
LUANNE 0
END

Figure 7-1.

record	name	pointer
1	LOIS	5
2	LOUISE	7
3	LARSON	8
4	LARRY	3
5	LOLA	2
6	LISA	1
7	LUANNE	0
8	LESLIE	6

This is shown schematically in Figure 7-2.

We will enter each record with 53 characters + 5 for the record pointer. We will keep separate lists for each letter of the alphabet to speed up the search process a little. This means that we must know where each of the possible 26 lists begins (we need a pointer to the start of each list). We also need to know where the next unused record is so that we know where to write a new addition.

```
initialize: get the start of each alphabetic list (one for
          each letter of the alphabet) and the start of
          the empty records

choice1: choose to select an existing number, add a new
          record, or quit

find an existing number:
          get the name whose number is desired,
          go to the start of the desired alphabetic list
          search through until a match is found
               if found,
                print the information
                goto choice2
               if not found,
                write appropriate message
                goto choice1

add a new record: get the data to be added
          go to the start of the desired alphabetic list
          search through until a name is found which is
               greater than the new name (follows it
               alphabetically)
          make new record point to this found record and
               make previous record point to new record
          change pointer of previous record
          write new data onto file

choice2: select choice1 or make a change

make a change: get data to be changed
               write changed data onto file

quit: write the new values of the pointers at record 0
      end
```

Let's begin to write the program using this program structure.

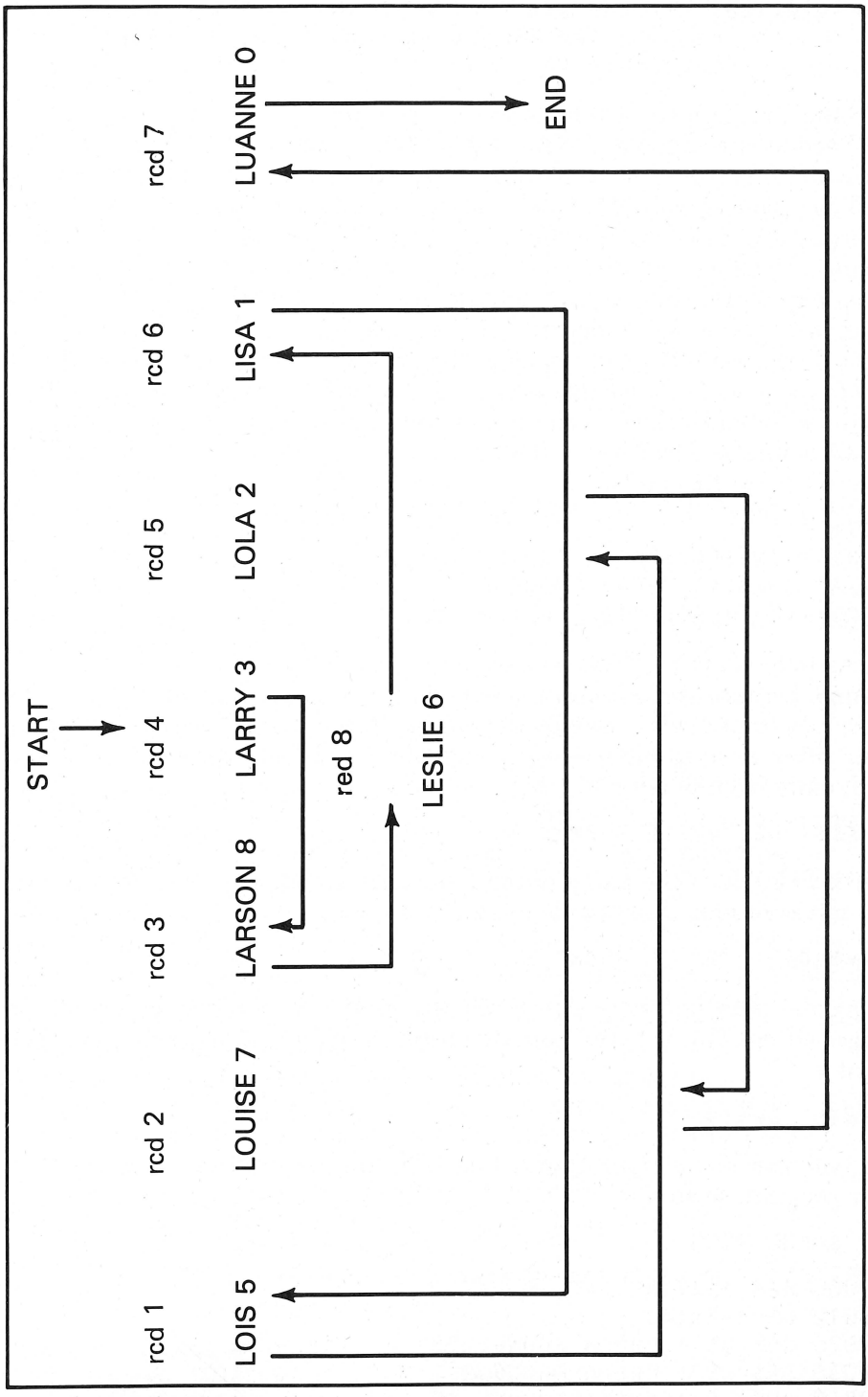

START
rcd 1
rcd 2
rcd 3
rcd 4
rcd 5
rcd 6
rcd 7
LOIS 5
LOUISE 7
LARSON 8
LARRY 3
LOLA 2
LISA 1
LUANNE 0
END
red 8
LESLIE 6

Figure 7-2.

```
initialize: get the start of each alphabetic list (one for
          each letter of the alphabet) and the start of
          the empty records
```

In this program we will maintain 26 separate lists of names and numbers. We need to have a pointer to the top of each of these lists because we need to know the first record in which data is stored for that letter of the alphabet.

These pointers to the top of each of the alphabetic lists can be stored in records 0, 1, and 2 of the file. Because we will be using them frequently during execution, we can save time by reading them into RAM when we begin execution. An easy way to do this is to read them into an array called T for top, using the 0 element of the T array to point to the first empty record. Because we will be reading the values for the pointers one after the other, we need only tell the computer to start reading at record 0. It will continue to read values, each one separated by a delimeter, until it has read all 27 pointers. This means it will read into records 1 and 2. The BASIC statements for this are

```
DIM T(26)
PRINT CHR$(4);"OPEN PHONE, L58"
READ CHR$(4);"READ PHONE,R0"
FOR I=0 TO 26:INPUT T(I):NEXT I
```

These statements will have to have line numbers in the program.

Also, because we will be working with the file so frequently, we can save ourselves some typing time by defining strings equal to the things we would most often have to type. If we define D$ as CHR$(4), for example, we will frequently have to type in

```
PRINT D$;"WRITE PHONE, R";N
```

where PHONE is the name of the file and N stands for the record number. We can save some time by defining

```
DW$=D$ + "WRITE PHONE,R"
```

Using the plus sign with strings merely joins the strings together into one longer string. This is called **concatenation**. Now we can write the same statement as

```
PRINT DW$;N
```

We will start our initialization at line 9000 to get it out of the way. Our program section is

```
5 GOSUB 9000

9000 REM INITIALIZATION
9010 D$=CHR$(4)
9020 DO$=D$ + "OPEN PHONE,L58"
9030 DR$=D$ + "READ PHONE,R"
9040 DW$=D$ + "WRITE PHONE,R"
9050 DC$=D$ + "CLOSE PHONE"
9060 DIM T(26):PRINT DO$
```

```
9070 PRINT DR$,0
9080 FOR I=0 TO 26: INPUT T(I): NEXT I
9090 PRINT DC$
9100 RETURN
```

The next section of the program requires a menu selection. Again we need only modify the menu program of Section 5.5.

```
choice1: choose to select an existing number or add a new
         record.

10 HOME
20 PRINT "CHOOSE ONE OF THE FOLLOWING:"
30 PRINT
40 PRINT TAB(5);"1 -- FIND A PHONE NUMBER"
50 PRINT TAB(5);"2 -- ADD A PHONE NUMBER"
60 PRINT TAB(5);"3 -- QUIT"
80 PRINT: PRINT "TYPE IN ONE NUMBER ";: GET A: PRINT A
90 ON A GOTO 1000, 3000, 10000
100 FLASH: VTAB(15): PRINT TAB(5);"ONLY VALUES FROM 1 TO 3
     ALLOWED": NORMAL
110 FOR K=1 TO 2000: NEXT K
120 VTAB(15): PRINT SPC(40)
130 VTAB(8): HTAB(20): PRINT " ": VTAB(8): HTAB(20): GET A:
     PRINT A: GOTO 90
```

The menu as written implies that the routine for finding an existing number will be located starting at line 1000 (this is option 1).

To associate names beginning with a particular letter with the correct pointer, we will use a function which works opposite to the CHR$( ) function you learned in Section 7.1. This is the **ASC( )** function. It returns the ASCII value of the **first letter** of the string which is in parentheses. Thus, ASC("LOIS") will equal 76 because that is the ASCII value for the letter L. Check the table of ASCII values in the appendix to this chapter to verify this.

If we read in a name which begins with A, we will find its ASCII value (65), and then subtract a 64 from it. This leaves us with the value 1, and indeed the pointer to the first phone number in the A list will be found in T(1).

```
find an existing number:
          get the name whose number is desired,
          go to the start of the desired alphabetic list
          search through until a match is found
               if found,
                print the information
                goto choice2
               if not found,
                write appropriate message
                goto choice1

1000 REM FIND A NUMBER
1010 INPUT "WHAT IS THE LAST NAME? ";Q$
1020 PRINT DO$
1030 I=ASC(Q$) - 64
1040 N=T(I)
```

```
1050 PRINT DR$;N
1060 INPUT L$,F$,N$,P
1070 IF Q$=L$ THEN 1200
1080 IF P <> 0 THEN N=P: GOTO 1050
1090 PRINT "SORRY, THIS NAME IS NOT ON FILE"
1100 PRINT DC$:GOTO 10
1200 HOME : HTAB(3): PRINT TAB(12);L$;SPC(3);F$
1210 PRINT TAB(12);N$
1220 PRINT DC$: GOTO 1500
```

In this section we search for the correct place. When we find it, N will be the record number where the new listing will be added:

```
add a new record: get the data to be added
          go to the start of the desired alphabetic list
          search through until a name is found which is
               greater than the new name (follows it
               alphabetically)
          make new record point to this found record and
               make previous record point to new record
          change pointer of previous record
          write new data onto file
```

```
3000 ADD A NEW RECORD
3010 HOME: VTAB(5): INPUT "WHAT IS THE LAST NAME? ";LN$
3020 INPUT "WHAT IS THE FIRST NAME? ";FN$
3030 INPUT "WHAT IS THE PHONE NUMBER? ";NN$
3040 FLASH :PRINT "PLEASE WAIT":NORMAL
3050 I=ASC(LN$) - 63: N=T(I): NA=T(0): T(0)=T(0)+1
3050 N1=0: PRINT DO$
3060 IF N=0 THEN GOTO 3100
3070 PRINT DR$;N
3080 INPUT L$,F$,N$,P
3090 IF LN$ > L$ THEN N1=N: N=P: GOTO 3070
3100 PN=N: PRINT DW$;NA:REM PREPARE TO WRITE NEW RECORD
3110 PRINT LN$;"," ;FN$;",";NN$;",";PN
3120 IF N1=0 THEN T(I)=NA:GOTO 3200
3130 PRINT DR$,N1
3130 INPUT L$,F$,N$,P
3140 PRINT DW$,N1
3150 PRINT L$;",";F$;",";N$;",";NA
3200 PRINT DC$
3210 GOTO 10
```

The next section of the program also requires a modification to the menu. In this case, we do not want to clear the screen because we want the phone number to remain visible.

```
choice2: select choice1 or make a change
```

```
1500 REM SECOND MENU
1510 HTAB(10)
1520 PRINT "CHOOSE ONE OF THE FOLLOWING:"
1530 PRINT
```

```
1540 PRINT TAB(5);"1--RETURN TO MAIN MENU"
1550 PRINT TAB(5);"2--ALTER THIS INFORMATION"
1580 PRINT: PRINT "TYPE IN ONE NUMBER ";: GET A: PRINT A
1590 ON A GOTO 10, 5000
1600 FLASH: VTAB(15): PRINT TAB(5);"ONLY VALUES FROM 1 TO 2
     ALLOWED": NORMAL
1610 FOR K=1 TO 2000: NEXT K
1620 VTAB(15): PRINT SPC(40)
1630 VTAB(8): HTAB(20): PRINT " ": VTAB(8): HTAB(20): GET
     A: PRINT A: GOTO 1590
```

If we choose menu selection 2 in the above code, we are choosing to change the contents of record N. We ask for the new number and then rewrite this information into the file.

```
make a change:  get data to be changed
                write changed data onto file
```

```
5000 REM CHANGE PHONE NUMBER
5010 INPUT "WHAT IS THE NEW PHONE NUMBER? ";N$
5020 PRINT DO$
5030 PRINT DW$;N
5040 PRINT L$;",";F$;",";N$;",";P
5050 PRINT DC$
5060 PRINT "THE CHANGE HAS BEEN MADE"
5070 FOR I=1 TO 2000:NEXT I
5080 GOTO 10
```

Finally, we must write the new values of the pointers into the file before we quit. After this is done we can end the program.

```
quit:  write the new values of the pointers at record 0
       end
10000 REM QUIT ROUTINE
10010 PRINT DO$
10020 PRINT DW$;0
10030 FOR I=0 TO 26:PRINT T(I):NEXT I
10040 PRINT DC$
10050 END
```

Here is the complete program.

```
5 GOSUB 9000
10 HOME
20 PRINT "CHOOSE ONE OF THE FOLLOWING:"
30 PRINT
40 PRINT TAB(5);"1--FIND A PHONE NUMBER"
50 PRINT TAB(5);"2--ADD A PHONE NUMBER"
60 PRINT TAB(5);"3--QUIT"
80 PRINT: PRINT "TYPE IN ONE NUMBER ";: GET A: PRINT A
90 ON A GOTO 1000, 3000, 10000
100 FLASH: VTAB(15): PRINT TAB(5);"ONLY VALUES FROM 1 TO 3
     ALLOWED": NORMAL
110 FOR K=1 TO 2000: NEXT K
120 VTAB(15): PRINT SPC(40)
```

```
130 VTAB(8):  HTAB(20): PRINT " ": VTAB(8): HTAB(20): GET
    A: PRINT A: GOTO 90
1000 REM FIND A NUMBER
1010 INPUT "WHAT IS THE LAST NAME? ";Q$
1020 PRINT DO$
1030 I=ASC(Q$) - 64
1040 N=T(I)
1050 PRINT DR$;N
1060 INPUT L$,F$,N$,P
1070 IF Q$=L$ THEN 1200
1080 IF P <> 0 THEN N=P: GOTO 1050
1090 PRINT "SORRY, THIS NAME IS NOT ON FILE"
1100 PRINT DC$:GOTO 10
1200 HOME : HTAB(3): PRINT TAB(12);L$;SPC(3);F$
1210 PRINT TAB(12);N$
1220 PRINT DC$: GOTO 1500
1500 REM SECOND MENU
1510 HTAB(10)
1520 PRINT "CHOOSE ONE OF THE FOLLOWING:"
1530 PRINT
1540 PRINT TAB(5);"1--RETURN TO MAIN MENU"
1550 PRINT TAB(5);"2--ALTER THIS INFORMATION"
1580 PRINT: PRINT "TYPE IN ONE NUMBER ";: GET A: PRINT A
1590 ON A GOTO 10, 5000
1600 FLASH: VTAB(15): PRINT TAB(5);"ONLY VALUES FROM 1 TO 2
    ALLOWED": NORMAL
1610 FOR K=1 TO 2000: NEXT K
1620 VTAB(15): PRINT SPC(40)
1630 VTAB(8):  HTAB(20): PRINT " ": VTAB(8): HTAB(20): GET
    A: PRINT A: GOTO 1590
3000 ADD A NEW RECORD
3010 HOME: VTAB(5): INPUT "WHAT IS THE LAST NAME? ";LN$
3020 INPUT "WHAT IS THE FIRST NAME? ";FN$
3030 INPUT "WHAT IS THE PHONE NUMBER? ";NN$
3040 FLASH :PRINT "PLEASE WAIT":NORMAL
3050 I=ASC(LN$) - 63: N=T(I): NA=T(0): T(0)=T(0)+1
3050 N1=0: PRINT DO$
3060 IF N=0 THEN GOTO 3100
3070 PRINT DR$;N
3080 INPUT L$,F$,N$,P
3090 IF LN$ > L$ THEN N1=N: N=P: GOTO 3070
3100 PN=N: PRINT DW$;NA:REM PREPARE TO WRITE NEW RECORD
3110 PRINT LN$;"," ;FN$;",";NN$;",";PN
3120 IF N1=0 THEN T(I)=NA:GOTO 3200
3130 PRINT DR$,N1
3130 INPUT L$,F$,N$,P
3140 PRINT DW$,N1
3150 PRINT L$;",";F$;",";N$;",";NA
3200 PRINT DC$
3210 GOTO 10
5000  REM CHANGE PHONE NUMBER
5010 INPUT "WHAT IS THE NEW PHONE NUMBER? ";N$
```

```
5020 PRINT DO$
5030 PRINT DW$;N
5040 PRINT L$;",";F$;",";N$;",";P
5050 PRINT DC$
5060 PRINT "THE CHANGE HAS BEEN MADE"
5070 FOR I=1 TO 2000:NEXT I
5080 GOTO 10
9000 REM INITIALIZATION
9010 D$=CHR$(4)
9020 DO$=D$ + "OPEN PHONE,L58"
9030 DR$=D$ + "READ PHONE,R"
9040 DW$=D$ + "WRITE PHONE,R"
9050 DC$=D$ + "CLOSE PHONE"
9060 DIM T(26):PRINT DO$
9070 PRINT DR$,0
9080 FOR I=0 TO 26: INPUT T(I): NEXT I
9090 PRINT DC$
9100 RETURN
10000 REM QUIT ROUTINE
10010 PRINT DO$
10020 PRINT DW$;0
10030 FOR I=0 TO 26:PRINT T(I):NEXT I
10040 PRINT DC$
10050 END
```

Before we can run this program for the first time, we need to set up up the random file with the initial values of the pointers. The empty file starts at record 3. The pointers for the other files are 0 because they are empty. The program to set up the phone file is

```
5 D$=CHR$(4)
10 PRINT D$;"OPEN PHONE,L58"
20 PRINT D$;"WRITE PHONE,RO"
30 PRINT 3
40 FOR I=1 TO 26:PRINT 0:NEXT I
50 PRINT D$;"CLOSE PHONE"
60 END
```

7.7 Application 7-4: An Artificial Intelligence Game

In this section we will develop the structure for a game in which the computer appears to be learning from its mistakes. The computer asks a question on some topic to try and guess what you are thinking. The questions must be such that only yes and no answers are permissible. When the computer 'thinks' it knows the answer, it takes a guess. If the guess is correct, the computer wins; if not, you must teach it about the answer which you had in mind.

We will structure this so that you can adapt it to your favorite topics, whether you wish to develop a game on the states of the nation, famous

athletes, flowers, or television shows. In fact, this idea is very similar to the diagnostic tests currently being developed for computers to help doctors correctly diagnose illnesses from the symptoms presented.

The program will make use of random access files with pointers. Each record in the file will contain either a question or an answer. There will also be two pointers in each record. If the record contains a question, the first pointer will indicate the record with the next question to ask (or guess to make) if the answer to the question is yes; if the answer to the question is no, the second pointer will be used to point to the next question (or guess). If the record contains a guess rather than a question, both pointers will be set to -1.

For example, let's assume that you are developing the game to guess the states of the nation. At the start the computer 'knows' only two states: Florida and California. It knows California is on the West coast and Florida is not. The file structure might look like this:

record	question (answer)	yes ptr	no ptr
1	Is it on the West coast?	2	3
2	California	-1	-1
3	Florida	-1	-1

The computer begins by asking the first question:

```
Is it on the West coast?
```

Suppose you are thinking of California. You would answer yes. The yes pointer would indicate that the next record to look at is record 2. When the computer looks at record 2, it sees that this is an answer record because the pointers are -1. Thus, it might say something like:

```
Is the answer California?
```

You would answer yes and the computer would win.

Similarly, if you had been thinking of Florida, then you would answer no to the question. The no pointer would lead to the answer at record 3, and the computer would respond with something like:

```
Is the answer Florida?
```

Again you would say yes and the computer would win.

Now suppose you were thinking of Kansas. When the computer asked its question, you would respond no. This would again lead to a guess of Florida, but this time the computer would be wrong and you would win. At this point, you would want to teach the computer something about Kansas.

To do this the computer will ask you what the correct answer is and ask for a new question. It will say something like:

```
Give me a question which will distinguish Kansas from
Florida such that the correct answer for Florida is yes.
```

You might reply with a question like:

```
Is it surrounded on three sides by water?
```

The computer will now enter this new question and the new answer into its file so that next time it can correctly distinguish them. The new file will look like this:

```
record  question (answer)              yes ptr  no ptr
  1     Is it on the West coast?           2       4
  2     California                        -1      -1
  3     Florida                           -1      -1
  4     Is it surrounded on three          3       5
          sides by water
  5     Kansas                            -1      -1
```

Note we had to add the two new records with the appropriate pointers and change the no pointer on the question in record 1. Now when the computer asks whether the the state is on the West coast, a no answer will point to record 4 which has the new question.

We begin by initializing, finding out where the next empty record is, and setting the pointer to the current record at 1 (this assumes that the first question is at record 1).

```
initialize:  find number of answers now known (N),
          get pointer to next empty record (PE)
          set current record pointer to 1 (RC)

read:    read in question (Q$), yes pointer (PY),
         and no pointer (PN) from current record

question:  ask question (or give guess)
     if answer is yes,
          set next record pointer (RN) to yes pointer
          if RN=-1, guess is correct, play again or end
          else set yes answer indicator, change values of
               previous and current record pointer,
               goto read
     if answer is no,
          set next record pointer (RN) to no pointer
          if RN=-1, guess is wrong, goto learn
          else  set no answer indicator, change values of
               previous and current record pointer,
               goto read

learn: get correct answer
       get distinguishing question
       set  next record pointer to first empty record and
              reset empty record pointer
       write new answer at RN + 1 with -1 pointers
       write  new question at RN,
              yes pointer equals old answer which is at
              RC, no pointer equals new answer which is at
              RN+1
       change  appropriate pointer in previous record (RP)
              to RN according to whether yes or no
              indicator is on
```

```
        rewrite new number of answers known, new pointer to
              empty space

play again or end: see if new game is desired.
        if yes, goto initialize
        if no, then end
```

The completed program is given below. The file is called AIFILE and has a length of 67 bytes or characters: bytes 0-59 hold the question or guess; 60-63 hold the yes pointer; 64-67 hold the no pointer. The first question is at record 1. Record 0 contains the number of answers known and the pointer to the next empty record.
Variables used are:

Q$	question (or guess)
RC	the current record being read
RP	the previous record read
RN	the next record to be read
PE	the next empty record
B	yes/no indicator (=60 if yes, =64 if no), corresponds to byte at which appropriate pointer is located
N	number of answers known by computer

The following program lines correspond to sections of the solution structure:

solution	lines
initialize	10,1000-1060
read	100
question	105-170
learn	500-610
play again or end	20-30

Here is the program.

```
10  GOSUB 1000: GOTO 100
20  INPUT "WANT TO PLAY AGAIN (Y/N)? ";A$
30  IF A$="N" THEN END
40  GOSUB 1000
100  PRINT DR$;RC: INPUT Q$: PRINT DR$;RC;",B60": INPUT
          PY: PRINT DR$;RC;",B64": INPUT PN: PRINT D$
105  IF VAL (PY)=- 1 THEN PRINT "IS THE ANSWER ";
110  PRINT Q$;" (Y/N) ": INPUT A$
115  IF A$="N" THEN RN=VAL (PN): GOTO 160
120  IF A$="Y" THEN RN=VAL (PY)
130  IF RN < > - 1 THEN B=60:RP=RC:RC=RN: GOTO 100
140  IF RN=- 1 THEN PRINT "CORRECT": GOTO 20
160  IF RN < > - 1 THEN B=64:RP=RC:RC=RN: GOTO 100
170  IF RN=- 1 THEN PRINT "I GIVE UP ": GOSUB 500
500  REM LEARNING SOMETHING NEW
510  PRINT "WHAT IS THE CORRECT ANSWER? ";QA$
520  PRINT "GIVE ME A QUESTION WHICH WILL DISTINGUISH"
530  PRINT Q$;" FROM ";QA$;"SO THAT THE"
540  PRINT "ANSWER FOR ";Q$;" IS YES"
```

```
560  INPUT Q$:RN=PE:PE=PE + 2
570  PRINT DW$;RN + 1: PRINT QA$: PRINT DW$;RN + 1;",B60":
          PRINT - 1: PRINT DW$;RN + 1;",B64": PRINT - 1
580  PRINT DW$;RN: PRINT Q$: PRINT DW$;RN;",B60": PRINT RN
          + 1: PRINT DW$;RN;",B64": PRINT RC
590  PRINT DW$;RP;",B";B: PRINT RN
600  PRINT DW$;0: PRINT N + 1;",";PE: PRINT DC$
610  RETURN
1000  REM INITIALIZATION
1005 R2=1:D$=CHR$ (4)
1010 DO$=D$ + "OPEN AIFILE,L67"
1020 DR$=D$ + "READ AIFILE,R"
1030 DW$=D$ + "WRITE AIFILE,R"
1040 DC$=D$ + "CLOSE AIFILE"
1050  PRINT DO$: PRINT DR$;0: INPUT N,PE
1060  RETURN
```

Again we must set up the AIFILE initially in order to begin the program. At the start we need to input the first question, the correct answer if the response to the question is "yes" and the correct answer if the response to the question is "no." In other words, we begin by making sure that the computer knows two correct answers. The first question will be stored at record 1 and will contain a pointer to the "yes" answer at record 2 and a pointer to the "no" answer at record 3. This means the pointer values must be written at bytes 60 and 64, respectively.

Record two will contain the "yes" answer as well as pointer values of -1 and -1. Record three will contain the "no" answer and -1 pointers. Record 0 will contain the value of N, the total number of answers known (2) and the value of PE, the pointer to the first empty record (4). Here is the set-up program.

```
10  GOSUB 1000
100  HOME : INPUT "FIRST QUESTION? ";Q$
110  INPUT "CORRECT ANSWER FOR YES? ";A1$
120  INPUT "CORRECT ANSWER FOR NO? ";A2$
500  PRINT DW$;1: PRINT Q$: PRINT DW$;1;",B60": PRINT 2:
          PRINT DW$;1;",B64": PRINT 3
510  PRINT DW$;2: PRINT A1$: PRINT DW$;2;",B60": PRINT -1:
          PRINT DW$;2;",B64": PRINT - 1
520  PRINT DW$;3: PRINT A2$: PRINT DW$;3;",B60": PRINT -1:
          PRINT DW$;3;",B64": PRINT - 1
530  PRINT DC$
540  END
1000  REM INITIALIZATION
1010 D$=CHR$ (4):DO$=D$ + "OPEN AIFILE,L67"
1020 DR$=D$ + "READ AIFILE,R"
1030 DW$=D$ + "WRITE AIFILE,R"
1040 DC$=D$ + "CLOSE AIFILE"
1050  PRINT DO$: PRINT DW$;0: PRINT 1,",";4: PRINT D$
1060  RETURN
```

Enter and run the set-up program. Enter the question 'Is it on the West coast' as the first question. Type in 'California' as the yes answer and 'Flor-

ida' as the no answer. You may wish to rename the file STATE instead of AIFILE in your programs. You can do this simply by changing the file name in lines 1010 through 1040 in both programs.

After you have initialized the file, try running the main program and teaching the computer some new states. Then go back and be sure you understand what the program is doing. When you feel confident you understand the program, use it to teach the computer about other things of interest to you. Remember that you should alter the file name for each different game so that you do not destroy previous files of information which the computer "knows."

7.8 Appendix—ASCII Codes

Each keyboard character is assigned a number between 0 and 255. This code number is called the **ASCII code** of the character.

For example, the letter "A" corresponds to the number 65, while the letter "Y" corresponds to the number 89. Punctuation marks and other keyboard characters also have ASCII code numbers; for instance, 40 corresponds to the symbol "(" and 62 corresponds to the symbol ">". Even the various control keys have corresponding codes; the space bar corresponds to the number 32 and the RETURN key to number 13. Table 7-1 lists the characters and their corresponding ASCII codes.

The Apple II keyboard is not capable of generating the symbols corresponding to all the ASCII codes. For example, in the Apple II Plus there are no keys to generate lower case letters, and you will not be able to see these characters on your screen. However, you may specify any ASCII symbols in your program by using the appropriate code numbers. This means you can insert lower case letters in a diskette file or send lower case letters to the printer.

We have already used the ASCII codes in a couple of places. In this chapter we used CHR$(4) to represent the CONTROL-D sequence because 4 was the ASCII code for CONTROL-D. We also used the ASCII values to help in locating the correct pointer for our alphabetical file in the random phone directory problem.

Table 7-1. ASCII character codes.

ASCII Code	*Character*	*ASCII Code*	*Character*
0	CONTROL-@	9	CONTROL-I
1	CONTROL-A	10	CONTROL-J
2	CONTROL-B	11	CONTROL-K
3	CONTROL-C	12	CONTROL-L
4	CONTROL-D	13	CONTROL-M
5	CONTROL-E	14	CONTROL-N
6	CONTROL-F	15	CONTROL-O
7	CONTROL-G	16	CONTROL-P
8	CONTROL-H	17	CONTROL-Q

Table 7-1. ASCII character codes (continued).

ASCII Code	*Character*	*ASCII Code*	*Character*
18	CONTROL-R	67	C
19	CONTROL-S	68	D
20	CONTROL-T	69	E
21	CONTROL-U	70	F
22	CONTROL-V	71	G
23	CONTROL-W	72	H
24	CONTROL-X	73	I
25	CONTROL-Y	74	J
26	CONTROL-Z	75	K
27	ESC	76	L
28	n/a	77	M
29	CONTROL-SHIFT-M	78	N
30	CONTROL-^	79	O
31	n/a	80	P
32	blank	81	Q
33	!	82	R
34	"	83	S
35	#	84	T
36	$	85	U
37	%	86	V
38	&	87	W
39	'	88	X
40	(	89	Y
41	)	90	Z
42	*	91	[
43	+	92	n/a
44	,	93	]
45	-	94	^
46	.	95	n/a
47	/	96	`
48	0	97	a
49	1	98	b
50	2	99	c
51	3	100	d
52	4	101	e
53	5	102	f
54	6	103	g
55	7	104	h
56	8	105	i
57	9	106	j
58	:	107	k
59	;	108	l
60	<	109	m
61	=	110	n
62	>	111	o
63	?	112	p
64	@	113	q
65	A	114	r
66	B	115	s

Table 7-1. ASCII character codes (continued).

ASCII Code	*Character*	*ASCII Code*	*Character*
116	t	123	{
117	u	124	n/a
118	v	125	}
119	w	126	.
120	x		
121	y		
122	z		

The computer uses ASCII codes to refer to letters and control operations. Any file, whether program or data, may be reduced to a sequence of ASCII codes. Consider the following address:

John Jones
2 S. Broadway

As a sequence of ASCII codes, it would be stored as

74,111,104,110,32,74,111,110,101,115,13,10
50,32,83,46,32,66,114,111,,97,100,119,97,121,13

Note the spaces are included (numbers 32) as are the carriage returns (RETURN) at the end of each line (number 13). ASCII codes allow us to describe any piece of text generated by the keyboard. This includes all formatting instructions like spaces, carriage returns, upper and lower case letters, and so forth. Once a piece of text has been reduced to a sequence of ASCII codes, it may also be faithfully reproduced on the screen or on a printer. This is the fundamental principle in the design of word processors.

TEST YOUR UNDERSTANDING 1 (answer on page 205)

Write a sequence of ASCII codes which will reproduce this ad.

FOR SALE: Beagle puppies, Pedigreed.
8 weeks, $125.

You may refer to characters by their ASCII codes by using CHR$. For example, CHR$(74) is the character corresponding to ASCII code 74 (upper case J); CHR$(32) is the character corresponding to ASCII code 32 (space). The PRINT instruction may be used in connection with CHR$. The instruction

```
10 PRINT CHR$(74)
```

will display an upper case J in the first position of the first print field.

TEST YOUR UNDERSTANDING 2 (answer on page 205)

Write a program which will print the ad of Test Your Understanding 1.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1 and 2

1: The ASCII codes required are:

```
70,79,82,32,83,65,76,69,58,32,66,101,97,103,108,
101,32,112,117,112,112,105,101,115,44,32,80,101,
100,105,103,114,101,101,100,46,13,32,32,32,32,32,
56,32,119,101,101,107,115,44,32,36,49,50,53,46,13
```

2: We will read in each ASCII code, then use the CHR$ function to retrieve the appropriate character. In order to prevent an OUT OF DATA error, we will use 999, which is not a valid ASCII code, as a sentinel value. Here is the program.

```
10 READ A
20 IF A=999 THEN END
30 PRINT CHR$(A):GOTO 10
40 DATA 70,79,82,32,83,65,76,69,58,32,66,101,97,103,
     108,101,32,112,117,112,112,105,101,115,44,32,80,
     101,100,105,103,114,101,101,100,46,13,32,32,32,
     32,32,56,32,119,101,101,107,115,44,32,36,49,50,
     53,46,13,999
```

8
A Peek into Your Apple's Memory

8.1 Bits, Bytes, and Where to Find Them

In Chapter One we introduced the notion of computer memory or main storage. There are two kinds of main storage: RAM or random access memory, and ROM or read only memory. When we talk about the size of a machine, we are usually talking about the size of RAM, the number of storage locations available for user programs and data.

As we said in Chapter One, memory locations in the Apple consist of 8 binary digits or bits. A set of eight bits is called a **byte**. One byte is enough to store one character: one letter, special symbol, or numerical digit stored as a string.

The contents of each of these memory locations can be accessed simply by reading the contents of the location at a certain **address**. The memory is like a set of boxes with numbers on them, each number being the address of the box. If we know the box number, we can access the contents of that box, or in the case of RAM, place new contents in it.

When we write programs in BASIC and give our variables names, the computer automatically associates a particular memory location with each variable name. The computer keeps a list of which address to go to when we refer to a specific variable by name. This is what makes programming in BASIC and in other programming languages so much more convenient than machine language.

The computer does not store the instructions and values in exactly the same way that we type them in. As we learned in Chapter Seven, the data is stored in the computer as numeric codes which consist only of 0's and 1's. The ASCII code for the character "A" is 65, but internally the computer stores this as

```
01000001
```

Each one of these 0's and 1's is called a **bit**. The word comes from **bi**nary digi**t**, the binary meaning only two.

Each memory location in a computer can store a certain number of these bits. In the Apple eight bits can be stored at each address. Thus, the Apple II

is sometimes called an eight-bit machine. Microcomputers are usually either 8 or 16-bit machines. Larger computers can store many more bits at a given location.

In the Apple II Plus and the Apple IIe, there is a maximum of 65,536 addressable memory locations. This much memory is always available with the IIe but must be added to the II Plus. This number represents the maximum amount of memory we can readily access provided we have a sufficient number of RAM chips installed. In a 48K (48 kilobyte) machine, there are only 48 * 1024 = 49,152 locations or bytes you can access. In a 64K machine, there are 65,536 available bytes. The symbol K, which usually stands for 1000 (kilo), represents 1024 in computer terminology. Because the computer stores only binary digits, it is customary to let the word kilo represent the power of 2 closest to 1000. Thus kilo is 1024, or 2 raised to the tenth power.

The addresses in the computer are numbered starting at location 0 up through the maximum number available less 1. In a 48K machine, accessible addresses range from 0 through 49151; in a 64K machine, addresses are 0 through 65535. The addresses can also be expressed as negative numbers by taking the actual address and subtracting the maximum allowable memory size from it. In an Apple II Plus, in which the maximum allowable memory size is 65536, we can represent the address 3072 as 3072-65536 = -62464. These two numbers, 3072 and -62464 refer to the same memory location. Location 0 has only one representation.

This is only a cursory introduction to some of the terminology and concepts you may encounter in other reading. As we continue through this chapter and discuss some advanced topics in the following chapters, we will use some of these concepts to take advantage of the capabilities of the Apple II.

8.2 PEEKs and POKEs

It is possible to write programs directly in bits and bytes. A program written this way is said to be written in machine language. Sometimes the only way to get your Apple to perform certain instructions is to write in machine language. We will want to use this in Chapter Eleven when we talk about producing sounds with the Apple.

Even if we do not write entire programs in machine language, there are some occasions when we wish to take a look at what is stored in certain memory locations or put some specific values into memory. We can do this with two statements in BASIC: the PEEK and the POKE commands.

The PEEK command fetches the contents of a specific memory location. Its format is

```
PEEK address
```

where address represents the address expressed in the decimal number system whose contents we wish to see.

The POKE command writes a number into some location. Its format is

```
POKE address, number
```

where address is the location into which we wish to place the value or number.

Setting the Screen Format

By using PEEKs and POKEs we can see what is happening on the screen and have some control over it. In Section 5.5 we discussed the HTAB and VTAB commands which moved the cursor to a designated spot on the screen. Sometimes we want to have the program look to see where the cursor is. We can do this by peeking at locations 36 and 37.

To find the current location of the cursor horizontally, we set some variable equal to the contents of memory location 36:

```
X=PEEK(36)
```

The resulting value of X is the horizontal column number from 0 to 39. To find out which row the cursor is in, we say

```
Y=PEEK(37)
```

The value of Y will be some number between 0 and 24, with row 0 at the top of the screen.

Normally, the screen as we view it is 40 columns and 24 rows. This is said to be the **text window**. Sometimes, we wish to change the size of the text window. We can use POKEs to do this.

To change the size of the text window, we need to POKE values for the new margins into particular memory locations.

These locations are:

parameter	location	permissible values	restrictions
left margin	32	0-39	
width	33	1-40	depends on left margin
top margin	34	0-23	must be above bottom margin
bottom margin	35	0-24	must be below top margin

The parameter values for the normal text screen are:

left margin	=	0
width	=	40
top margin	=	0
bottom margin	=	24

Changing the margins can be very helpful when you are editing using the cursor movement keys. If you have tried to edit a statement, you know that when the computer displays the line on the screen, it does not use all 40 columns across. This can cause some problems if you are trying to edit a PRINT statement and some literal constant is split between two lines.

For example, if you type in the following statement:

```
10 PRINT "THIS IS A TEST TO SEE THE TEXT WINDOW COMMANDS
   WORK"
```

and then ask the computer to LIST the line, it will appear as follows on your 40-column screen:

```
10 PRINT "THIS IS A TEST TO SEE
      THE TEXT WINDOW COMMANDS WOR
      K"
```

If you try to move the cursor along this line from the beginning to the end in order to "copy" it, the blanks which the computer has inserted for readability while it presents the line on the screen will be embedded in your literal constant. When you RUN the program you will obtain

```
THIS IS A TEST TO SEE          THE TE
XT WINDOW COMMANDS WOR            K
```

Obviously, this is not what you want.

To simplify your editing and to end your worry about this problem, you can begin by clearing the screen (HOME). Then type in

```
POKE 33,33
```

This sets the screen width to 33 columns. If you list the line now, you will see there are no extra spaces left. Thus, you can move the cursor from one end of the line to the other without embedding any extra spaces. Try this editing process with the line above. Note that you will not be able to move past column number 33 on the right of the screen.

To return to the normal screen dimensions simply type in

```
TEXT
```

This command can also be included in a BASIC program.

As another example of how the text window works, let's construct a program to write a title at the top of the screen and then have the catalog appear at the bottom. If you use your System Master disk, which has a long catalog, you will have a better idea of the meaning of a text window. Here is the program.

```
10 HOME
20 VTAB(5);PRINT "EXAMPLE OF A TEXT WINDOW"
30 POKE 34,10
40 PRINT CHR$(4);"CATALOG"
```

When you run this program, you will see that the catalog only appears in the lower part of the screen. The upper part remains untouched.

Event Trapping

Sometimes we want to look in to see if some event has occurred. Usually, these events have to do with seeing if the keyboard has been pressed. To find

out we look at memory location -16384. If it contains a value > 127, something was typed at the keyboard. We can use something like

```
X=PEEK(-16384)
20 IF X > 127 THEN PRINT "A KEY WAS HIT"
```

If it is necessary to know which key was hit, we can find out by subtracting 128 from the value of X. The resulting number will be the ASCII value of the key struck.

To reset the keyboard to detect when a key is struck again, we must POKE -16368,0.

There are many other useful locations that we will want to use in the pages to come. One helpful technique involving the use of PEEKs and POKEs is given in the next section.

8.3 Catching Errors Before They Cause Trouble

A well-written program is one that protects the user as much as possible from inadvertent errors. A good deal of this can be done within the usual BASIC statements by trying to anticipate possible problems. If they are anticipated, appropriate IF statements can often be used to compensate for the error before it causes trouble. For an example of this review the menu program we introduced in Section 5.5. Here we anticipated that the user might type in a wrong number, and we developed some logic to deal with it.

We did not anticipate the situation in which the user might type in a letter instead of a number. In this case the computer would simply continue to say

```
?REENTER
```

until a number was typed in. We could have trapped this error, or provided for this possibility, in a better way.

To do this we could read in a string instead of a number. Our statement would be

```
80 PRINT:PRINT "TYPE IN ONE NUMBER ";:GET A$:PRINT A$
```

We would then use the ASC function which we discussed in Chapter Seven to obtain the ASCII code of the number we input and, by subtracting 48 from that value, obtain an equivalent number. This is because the ASCII value of the number one is 49. Our statement would look like

```
85 A=ASC(A$) - 48
```

This would allow us to trap alphabetic input and use the same reminder about the allowable range of numbers for input.

Sometimes the nature of the error makes it difficult to protect the user without having some more powerful techniques at hand. Suppose, for example, we wanted to use sequential files and allow the user to append new records to a file. If the user inputs the name of a file which does not exist, an error will be returned when we try to append to it.

Try this program.

```
10 D$=CHR$(4)
20 INPUT "WHAT IS THE FILE NAME? ";A$
30 PRINT D$;"APPEND";A$
40 PRINT D$;"WRITE";A$
50 PRINT "ADD THIS RECORD"
60 PRINT D$;"CLOSE";A$
```

When you run it, you will obtain an error message that says

```
FILE NOT FOUND
BREAK IN 30
```

For the typical user of a program, this error message won't make much sense. The user also stands the chance of losing all the work done by the program so far, including any data which might have been input. A well-planned program will protect the user from these kinds of things.

We can do this by using the **ONERR GOTO** statement. The form of this statement is

```
ONERR GOTO line#
```

If it is placed before the lines at which we anticipate problems, any errors found in those lines will send control to the specified line number. At that line number we can have an error routine.

Try adding lines 15, 70, and 100 to the above program to obtain this new version.

```
10 D$=CHR$(4)
15 ONERR GOTO 100
20 INPUT "WHAT IS THE FILE NAME? ";A$
30 PRINT D$;"APPEND";A$
40 PRINT D$;"WRITE";A$
50 PRINT "ADD THIS RECORD"
60 PRINT D$;"CLOSE";A$
70 END
100 PRINT "THIS IS AN ERROR ROUTINE"
```

Now run the program. Instead of the operation simply terminating, the computer goes to line 100 to follow the instructions there. All we need now is to provide some means of dealing with the error starting at line 100.

First, we need to know what the error is. Some possibilities are: the user has requested an append to a file which does not exist; has tried to write data on a disk which is already full; has named a file which is not a text file; has failed to put the disk in the drive; or has forgotten to close the drive door.

We can discover precisely what the error is by PEEKing at location 222. The computer stores the code for the error which has occurred at that location. Possible error codes are:

Applesoft:

0	NEXT without FOR
16	Syntax
22	RETURN without GOSUB
42	OUT OF DATA

53	Illegal Quantity
69	Overflow
77	Out of memory
90	Undefined statement
107	Bad Subscript
120	Redimensioned array
133	Division by zero
163	Type mismatch
176	String too long
191	Formula too complex
224	Undefined function
254	Bad response to an input statement
255	Ctrl-C interrupt attempted

DOS:

1	Language not available
2,3	Range error
4	Write protected
5	End of data
6	File not found
7	Volume mismatch
8	I/O error
9	Disk full
10	File locked
11	Syntax error
12	No buffers available
13	File type mismatch
14	Program too large
15	Not direct command

Suppose in a given program we wish to protect against the three specific types of errors we believe to be most likely to occur. These are:

error 6:	tried to append to a file which does not exist
error 13:	named a file which is not a text file
error 8:	has failed to put the disk in the drive or close the drive door.

For any other error condition, we will merely print some appropriate message and end the program. Our routine in line 100 might look like this:

```
100 X=PEEK(222)
110 IF X=6 THEN GOTO 200
130 IF X=13 THEN GOTO 300
140 IF X=8 THEN GOTO 400
150  PRINT  "AN ERROR HAS OCCURRED.  ITS  ERROR  CODE
     NUMBER IS ";X
160  PRINT "SEE YOUR APPLESOFT OR DOS MANUAL"
170 END
200 REM FILE DOES NOT EXIST
210 PRINT "THE FILE YOU NAMED ;A$;" DOES NOT EXIST."
```

```
220 PRINT "DO YOU WISH TO "
230 PRINT:PRINT TAB(16);"1 -- PROVIDE A DIFFERENT NAME"
240 PRINT "OR";TAB(16);"2 -- START A NEW FILE"
250 INPUT "TYPE 1 OR 2 ";A
260 IF A=1 THEN GOTO 20
270 IF A=2 THEN PRINT D$;"OPEN";A$;GOTO  40
280 GOTO 250
300 REM NOT A TEXT FILE
310 PRINT "THE FILE YOU NAMED ;A$;" IS NOT A TEXT FILE"
320 PRINT "DO YOU WISH TO "
330 PRINT:PRINT TAB(16);"1 -- PROVIDE A DIFFERENT NAME"
340 PRINT "OR";TAB(16);"2 -- SEE THE CATALOG"
350 INPUT "TYPE 1 OR 2 ";A
360 IF A=1 THEN GOTO 20
370 IF A=2 THEN PRINT D$;"CATALOG":GOTO  40
380 GOTO 350
400 REM I/O ERROR
410 PRINT "THE FILE CANNOT BE ACCESSED BECAUSE  OF  A
     DISK PROBLEM"
420 PRINT  "PLEASE   CHECK  TO SEE  THAT  THE  DISK  IS
     PROPERLY INSERTED"
430 PRINT "AND THE DISK DRIVE DOOR IS CLOSED"
440 PRINT: PRINT "WHEN READY, HIT RETURN": GET A$
450 RESUME
```

The first two error routines send control back to a particular line number to continue the operation. The last routine ends with the command **RESUME** in line 450.

The command to RESUME sends control back to the same line the computer was executing when it ran into trouble. It can only be used in conjunction with the ONERR GOTO command.

The complete program looks like this:

```
10 D$=CHR$(4)
15 ONERR GOTO 100
20 INPUT "WHAT IS THE FILE NAME? ";A$
30 PRINT D$;"APPEND";A$
40 PRINT D$;"WRITE";A$
50 PRINT "ADD THIS RECORD"
60 PRINT D$;"CLOSE";A$
70 END
100 X=PEEK(222)
110 IF X=6 THEN GOTO 200
130 IF X=13 THEN GOTO 300
140 IF X=8 THEN GOTO 400
150  PRINT  "AN  ERROR HAS OCCURRED.   ITS  ERROR  CODE
     NUMBER IS ";X
160  PRINT "SEE YOUR APPLESOFT OR DOS MANUAL"
200 REM FILE DOES NOT EXIST
210 PRINT "THE FILE YOU NAMED ;A$;" DOES NOT EXIST."
220 PRINT "DO YOU WISH TO "
230 PRINT:PRINT TAB(16);"1 -- PROVIDE A DIFFERENT NAME"
```

```
240 PRINT "OR";TAB(16);"2 -- START A NEW FILE"
250 INPUT "TYPE 1 OR 2 ";A
260 IF A=1 THEN GOTO 20
270 IF A=2 THEN PRINT D$;"OPEN";A$;GOTO  40
280 GOTO 250
300 REM NOT A TEXT FILE
310 PRINT "THE FILE YOU NAMED ;A$;" IS NOT A TEXT FILE"
320 PRINT "DO YOU WISH TO "
330 PRINT:PRINT TAB(16);"1 -- PROVIDE A DIFFERENT NAME"
340 PRINT "OR";TAB(16);"2 -- SEE THE CATALOG"
350 INPUT "TYPE 1 OR 2 ";A
360 IF A=1 THEN GOTO 20
370 IF A=2 THEN PRINT D$;"CATALOG":GOTO 40
380 GOTO 350
400 REM I/O ERROR
410 PRINT  "THE FILE CANNOT BE ACCESSED BECAUSE  OF  A
     DISK PROBLEM"
420 PRINT  "PLEASE  CHECK  TO SEE  THAT  THE  DISK  IS
     PROPERLY INSERTED"
430 PRINT "AND THE DISK DRIVE DOOR IS CLOSED"
440 PRINT: PRINT "WHEN READY, HIT RETURN": GET A$
450 RESUME
```

In some cases it may be necessary to know which line number in the program caused the problem. A line number may be too large to be stored in only one memory location; in the Apple II it is stored in two adjacent memory locations, 218 and 219. A line number is also stored as a binary number. When we PEEK into these locations, the computer will return the decimal equivalent, but because there are two locations for the line number, we must assemble the two parts of the number.

The Apple II computer stores the high order digits in the second location and the low order digits in the first location. For example, if the error occurred at line number 999 in your program, the number 999 would be translated into a binary number first (binary numbers and their decimal equivalents are discussed later in this section). The binary equivalent of 999 is

```
00000011 11100111
```

The 00000011 or high order digits would be in location 219, and the 11100111 or low order digits would be in location 218.

To assemble the two parts correctly we would use

```
X=PEEK(219)*256 + PEEK(218)
```

X would have the value of the line number in decimal. We will discuss the logic behind this in the next section.

If you wanted the error routine to apply only to a particular section of your program, you could remove the ONERR GOTO condition by using

```
POKE 216,0
```

This permits the default error messages of the Apple II to take over. In some cases the ONERR GOTO will not operate correctly. This occurs when there

are two GET errors in a row or when there are repeated errors. See your Apple II User's Guide for details.

Binary, Decimal, and Hexadecimal Numbers

By this time you may be curious about the relationship between decimal numbers and binary numbers. In this section we will try to give you an idea of what this relationship is. If you would rather skip this section, you may do so. Simply following the instructions as given in the following sections will permit you to do some rather nice things without worrying about the whys and wherefores.

Before you can understand the binary number system, you must understand how the decimal number system works. The decimal number system is called a base ten system. There are ten digits which can be used: 0,1,2,3,4,5,6,7,8, and 9. The value a particular digit has in a number depends not only upon the digit itself but also upon the position of the digit in the number: 123 and 321 use the same digits but have different values.

The value of the lowest order digit is the value of that digit times 10 raised to the zero power. Because any number raised to the zero power is equal to one, we say that the lowest order position is the ones position. Similarly, the next-to-last position is the tens position, or ten raised to the first power. The third from the last position is the hundreds position (ten raised to the second power). Each time we move a digit one position further out its value is multiplied by ten. Thus, 123 is one hundred, two tens, and three ones; 321 is three hundreds, two tens and one one.

If we have a base two number system, a binary system, then there are two possible digits: 0 and 1. If we wish to translate a binary number into decimal, we multiply each digit in successive positions by a higher power of two. Thus, to translate 11000111 into decimal we multiply the lowest digit by 1 (2 raised to the 0 power), the next lowest digit by 2 (2 raised to the 1 power), the third lowest by 4 (2 squared), and so on. Doing this shows us that

11000111 base two

equals, starting from the right:

$$1 \times 1 + 1 \times 2 + 1 \times 4 + 0 \times 8$$
$$+ 0 \times 16 + 0 \times 32 + 1 \times 64 + 1 \times 128 = 199 \text{ base ten.}$$

TEST YOUR UNDERSTANDING 1 (answers on page 219)

Convert the following binary numbers to decimal.

a. 11100111
b. 00111100
c. 10000001

You can construct a positional number system using any integer for the base, but there are only a few of any interest to us. One frequently used in dealing with computers is the ***hexadecimal*** system. This is a base 16 system. It has 16 digits, represented as

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

That is, we use letters to represent digits for 10, 11, 12, 13, 14, and 15.

The value of a hexadecimal number is that it can be used as a shortcut representation of a binary number. Each set of four binary digits translates into one hexadecimal digit as such:

binary	hexadecimal	decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

This means 1100 0111 in base two equals C7 in base sixteen.

To translate the strange-looking hexadecimal number into a decimal number we use powers of 16.

7 times 16 to the zero power
plus C which is 12, times 16 to the first power=
7x 1 + 12 × 16=231 in decimal.

If we have a binary number like

00000011 11100111

its hexadecimal representation is

03 E7

and the equivalent decimal representation is

0 × 4096 + 3 × 256 + 14 × 16 + 7 × 1=999.

The translation from binary to hexadecimal was very direct and did not require us to do any multiplication. Each set of four binary digits was translated. The translated digits were then placed next to each other. You will

encounter binary and hexadecimal numbers frequently if you get very involved in programming in machine language.

Let's return to our previous statement about memory locations 218 and 219 which contain the line number in our BASIC program where an error occurred. We said that in order to learn the correct statement number we had to assemble the two parts as

```
PEEK(219) * 256 + PEEK(218)
```

Let's say that our memory locations have the following values:

1001 1100	0001 0011
218	219

Because high order digits are in 219, the correct binary representation of the line number is:

0001 0011	1001 1100
219	218

In hexadecimal, this is:

1 3	9 C
219	218

Remember that C is 12 in decimal.

To find the correct decimal equivalent we must multiply each hexadecimal digit by the correct power of 16 and add them together. Thus, we have:

$$1 \times 4096 + 3 \times 256 + 9 \times 16 + 12 \times 1 = 5020$$

When we PEEK into a memory location, the computer returns the decimal equivalent of the contents of that memory location. Thus, by PEEKing 218 and 219 we would obtain

$1 \times 16 + 3 \times 1 = 19$ in location 219,

and

$9 \times 16 + 12 \times 1 = 156$ in location 218.

If we took each term in location 219 and multiplied it by 256, we would obtain

$$1 \times 4096 + 3 \times 256$$

which is exactly what we need to arrive at our value of 5020. We can translate each memory location separately and then multiply the high order value by 256 and add it to the low order value. In this way we obtain the number stored in these two consecutive locations.

TEST YOUR UNDERSTANDING 2 (answer on page 219)

Assume that memory locations 218 and 219 contain the following:

1100 0011	1000 0111
218	219

Find the correct line number in decimal that is stored in these locations.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1 and 2

1: a. 231
b. 60
c. 129

2: 195 is in location 218, and 135 is in 219. Thus, the correct line number is
$135 \times 256 + 195 = 34755$

9
Playing with Words

In Chapter Seven we introduced ASCII codes for characters and presented a table of the codes for printable characters. We also introduced the CHR$( ) function which, when a number was placed in the parentheses, would produce the character corresponding to that ASCII code; PRINT CHR$(65) caused an A to be printed.

In this chapter we will look at some other ways to manipulate strings in Applesoft. We will also look at one of the more important applications of these techniques: word processing.

9.1 Manipulating Strings

A string may consist of as many as 255 characters. If you try to type a string this long, you will find you cannot type more than 40 characters on a line. You may type strings containing more than 40 characters by continuing to type **without hitting the RETURN key**. When a line is filled, the computer will automatically place the next letter at the beginning of the next line. If you do not hit the RETURN key at the end of the line, the next line will be a continuation of the first (this is the wraparound feature discussed in Chapter Two).

There are a number of operations which may be performed on strings. First, strings may be "added" (or, in computer jargon, "concatenated"). Suppose that we have strings A$ and B$, with A$="WORD" and B$="PROCESSOR." The sum of A$ and B$, denoted A$ + B$, is the string obtained by joining A$ and B$

```
"WORDPROCESSOR"
```

Note that no space is left between the two strings. To include a space suppose that C$=" " or the string which consists of a single space. Then A$ + C$ + B$ is the string

```
"WORD PROCESSOR"
```

TEST YOUR UNDERSTANDING 1 (answer on page 226)

If A$="4" and B$="7", what is A$+B$?

The computer handles relations among strings in much the same way that it handles relations among numbers. For example, we say that the strings A$ and B$ are equal, written A$ = B$, provided that they consist of exactly the same characters in the same order. Otherwise the strings are unequal, written A$ <> B$ or A$ >< B$. The notation A$ < B$ means that A$ precedes B$ in alphabetical order. Similarly, A$ > B$ means that A$ succeeds B$ in alphabetical order. The following are true relations among strings:

"BEAR" < "GOAT"
"GIRL" > "BOY"

The notation A$ >= B$ means that either A$ > B$ or A$ = B$. Simply, this means that A$ either succeeds B$ in alphabetical order or A$ and B$ are the same. The notation A$ <= B$ has a similar meaning.

Example 1. Write a program which alphabetizes the following list of words: egg, celery, ball, bag, glove, coat, pants, suit, clover, weed, grass, cow, and chicken.

Solution. We set up a string array A$(J) which contains these 13 words. We set B$ equal to the first word. We successively compare B$ with each of the words in the array. If any compared word precedes B$, we replace B$ with that word. At the end of the comparisons, B$ contains the first word in alphabetical order. We place this as the first item in the array C$(J). We now repeat the process with the first word eliminated. This gives us the second word in alphabetical order and so forth. Here is our program.

```
10 DIM A$(13),D$(13),C$(13)
20 DATA EGG,CELERY,BALL,BAG,GLOVE,COAT
30 DATA PANTS,SUIT,CLOVER,WEED,GRASS
40 DATA COW,CHICKEN
50 FOR J=1 TO 13
60 READ A$(J):REM SET UP ARRAY A$
70 NEXT J
80 FOR K=1 TO 13:REM FIND KTH WORD IN ORDER
85 REM 90-200 CREATE AN ARRAY D$ CONSISTING OF THE
87 REM WORDS YET TO BE ALPHABETIZED
90 L=1
100 FOR J=1 TO 13
110 E=0
120 FOR M=1 TO 13
130 IF A$(J)=C$(M) THEN E=1
140 NEXT M
150 IF E=0 THEN 160
155 GOTO 200
160 D$(L)=A$(J)
170 L=L+1
200 NEXT J
210 B$=D$(1)
220 FOR L=1 TO 13-K+1
230 IF D$(L) < B$ THEN 250
235 GOTO 300
```

```
250 B$=D$(L)
300 NEXT L
400 C$(K)=B$
410 NEXT K
500 FOR K=1 TO 13
510 PRINT C$(K)
520 NEXT K
600 END
```

This program can be modified to make a program alphabetizing any collection of strings. We will leave the details to the exercises.

Functions Involving Strings

We learned about ASCII codes and the CHR$ function. This function is only one of many related to strings. To obtain the ASCII code of a character, use the instruction **ASC**. The instruction

```
20 PRINT ASC("B")
```

will print the ASCII code of the character "B," namely 66. In place of "B" you may use any string. The computer will return the ASCII code of the *first character* of the string. For example, the instruction

```
30 PRINT ASC(A$)
```

will print the ASCII code of the first character of the string A$. If A$ is the empty string " " (the string consisting of no characters), then using ASC(A$) will result in an ILLEGAL QUANTITY ERROR.

TEST YOUR UNDERSTANDING 2 (answer on page 226)

Determine the ASCII codes of the characters $, +, × by using the ASC function. Check your answers by looking at the chart in Section 7.8.

You may compute the length of a string by using the **LEN** instruction. For example, **LEN("BOUGHT")** is equal to six since the string "BOUGHT" has six letters. Similarly, if A$ is equal to the string "FAMILY INCOME", then **LEN(A$)** is equal to 13 (the space between the words counts!). Here is an application of the **LEN** instruction.

Example 2. Write a program that inputs the string A$ and then centers it on a line of the display.

Solution. A line is 40 characters long with the spaces numbered from 1 to 40. The string A$ takes up LEN(A$) of these spaces, so there are 40 - LEN(A$) blank spaces to be distributed on either side of A$. The line should begin with half of the 40-LEN(A$) spaces, or with (40-LEN(A$))/2 spaces, so we should tab to column (40-LEN(A$))/2 + 1. Here is our program.

```
10 INPUT A$
20 HOME
30 PRINT TAB((40-LEN(A$))/2 + 1); A$
40 END
```

TEST YOUR UNDERSTANDING 3

Use the program of Example 2 to center the string "THE APPLE II COMPUTER".

It is possible to dissect strings using the three instructions **LEFT$**, **RIGHT$**, and **MID$**. These instructions allow you to construct a string consisting of a specified number of characters taken from the left, right, or middle of a designated string. Consider the instruction

```
10 LET A$=LEFT$("LOVE",2)
```

The string A$ consists of the two leftmost characters of the string "LOVE". That is, A$="LO". The instructions

```
20 LET B$="TENNIS"
30 LET C$=RIGHT$(B$,3)
```

set C$ equal to the string consisting of the three rightmost letters of the string B$, namely C$="NIS". If A$="REPUBLICAN," the instruction

```
40 LET D$=MID$(A$,5,3)
```

sets D$ equal to the string consisting of three characters starting with the fifth character of A$, which is D$="BLI." These last three instructions are absolutely fundamental in the design of many programs. We will use these in some of the application programs in this chapter.

TEST YOUR UNDERSTANDING 4 (answer on page 226)

Determine the string constant

```
RIGHT$(LEFT$("COMPUTER",4),3)
```

When manipulating strings, it is important to recognize the difference between numerical data and string data. The number 14 is denoted by 14; the string consisting of the two characters 14 is denoted "14". The first is a numerical constant and the second a string constant. We can perform arithmetic using the numerical constants, but we cannot perform any of the character manipulation supplied by the instructions **RIGHT$**, **MID$**, and **LEFT$**. Such manipulation may only be performed on strings.

How may we perform character manipulation on numerical constants? BASIC provides a simple method. We first convert the numerical constants to string constants by using **STR$**. For example, the number 14 may be converted into the string "14" using the instruction

```
10 A$=STR$(14)
```

As a result of this instruction, A$ has the value "14". As another example, suppose that the variable B has the value 1.457. **STR$(B)** is then equal to the string "1.457".

To convert strings consisting of numbers into numerical constants, use VAL. Consider

```
20 B=VAL("3.78")
```

This instruction sets B equal to 3.78.

When we use the VAL function, the computer starts from the leftmost character of the string to convert to a number. It stops the conversion as soon as it hits a character which it cannot interpret as a number. The computer would return the following:

```
VAL("15-14")=15
VAL("+32")=32
VAL("25 LBS.")=25
VAL("HELLO")=0
```

When we introduced the idea of a menu, we indicated that we would want to protect the user from inputting a non-numeric character with the GET statement (see Section 5.5). We can do this quite easily using the VAL function. Instead of using the statement

```
GET A
```

to obtain the menu selection, use the following:

```
GET A$:A=VAL(A$)
```

Now, if the user types in a letter or plus or minus sign, the computer will accept it without stopping execution, but when the VAL function is executed, the resulting value will be out of range, causing the computer to again request input.

TEST YOUR UNDERSTANDING 5 (answer on page 226)

Suppose that A$ equals "5 PERCENT" and B$ equals "758.45 DOLLARS". Write a program that starts from A$ and B$ and computes five percent of $758.45.

Exercises (answers on page 321)

1. Use the program of Example 1 to alphabetize the following sequence of words: justify, center, proof, character, capitalize, search, replace, indent, store, and password.
2. Write a program to form the following string into lines consisting of 80 characters each:

Word processing will revolutionize the office. Already millions of word processing systems are in use. By the end of the decade, the typewriter will be totally obsolete. Word processing systems will increase in sophistication.

3. Write a program that rewrites the addition problem 15 + 48 + 97 = 160 in the form:

```
 15
 48
+97
160
```

4. Write a program that inputs the string constants "$6718.49" and "$4801.96" and calculates the sum of the given dollar amounts.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, 4, and 5

1: "47"

2:
```
10 DATA $,+,X
20 FOR J=1 TO 4
30 READ A$(J)
40 B(J)=ASC(A$(J))
50 PRINT A$(J), B(J)
60 NEXT J
70 END
```

4:
```
LEFT$("COMPUTER",4)="COMP"
RIGHT$("COMP",3)="OMP"
```

5:
```
10 A$="5 percent":B$="758.45 dollars"
20 A=VAL(A$):B=VAL(B$)
30 PRINT A$,"OF",B$,"IS"
40 PRINT A*B*.01
50 END
```

9.2 Application 9-1: A Word Guessing Game

This game is known as JOTTO. It is a variation of some popular games that require you to guess the sequence of colors your opponent has chosen. The game will be designed for two people. Each person thinks of a five-letter word in which no letter is repeated. The object of the game is to figure out your opponent's word before he figures out what word you are thinking of.

Each person writes down his word. The game begins with person 1 guessing a five letter word (no repeats). Person 2 then must tell how many letters in the guess are also in his chosen word. For example, if person 2 has written down the word "CAMEL" and person 1 guesses "QUICK", then person 2 must respond by saying that there is 1 letter in common. The game continues

in this manner with the two players taking turns until one person arrives at the correct answer.

The game requires using logic to eliminate letters and then trying to figure out the word from those that remain. It is a lot of fun, but its weakness is that you must correctly count the number of common letters. Even though this sounds trivial, it is a source of frequent error and frustration.

We will write a program which will let the computer figure out the count for us. At the start, each player will type the selected word into the computer. We will do this without letting the word be displayed on the screen. This will decrease the chance of your opponent accidentally seeing your word, but you will want to have your opponent turn away as you type it in so he cannot read the keyboard either.

Then the computer will prompt each person in turn for a guess and report the count of common letters. The computer will also announce the winner.

Here is the solution structure.

```
initialize:
     loop for each player
          get player's name
          get player's word character by character
     endloop

loop until word is guessed
     guess:
          loop for each player
               get player's guess
               if guess=opponent's word then goto gameover
               if guess is not correct, count letters in
               common
               report the letter count
          endloop
endloop

gameover: report winning word
     report losing word
```

Here is the program. Note the use of the GET statement to read the words in character by character. The characters are then concatenated into a single word. A doubly subscripted array is used to store the letters and word for each player.

```
10  HOME
15  INPUT "NAME OF FIRST PLAYER? ";N$(1)
20  INPUT "NAME OF SECOND PLAYER? ";N$(2)
30  VTAB (5): PRINT "IN THIS GAME EACH PLAYER WHISPERS A"
40  PRINT "FIVE-LETTER WORD WITH NO REPEATING"
50  PRINT "LETTERS TO ME. THE OTHER PERSON WILL"
60  PRINT "TRY TO ARRIVE AT YOUR WORD BY GUESSING"
70  PRINT "ANOTHER FIVE-LETTER WORD. I WILL TELL"
80  PRINT "THE PERSON HOW MANY LETTERS IN THE"
90  PRINT "GUESS ARE THE SAME AS IN YOUR WORD."
100  PRINT "HIT RETURN WHEN READY TO PROCEED": GET A$
```

```
110  HOME : FOR I=1 TO 2: POKE - 16368,0
120  PRINT N$(I): PRINT " PLEASE TYPE YOUR WORD IN
       CAREFULLY"
130  FOR J=1 TO 5
140  GET W$(I,J):W$(I,0)=W$(I,0) + W$(I,J)
150  NEXT J: PRINT "THANK YOU"
160  PRINT "HIT RETURN TO CONTINUE ": GET A$: HOME :
       NEXT I
200  FOR I=1 TO 2: POKE - 16368,0
210  HOME : PRINT N$(I): INPUT "WHAT IS YOUR GUESS?"
       ;G$(0)
220  FOR J=1 TO 5
230 G$(J)=MID$ (G$(0),J,1)
250  NEXT J
260 K=I + 1: IF K=3 THEN K=1
270  PRINT : IF G$(0)=W$(K,0) THEN GOTO 400
```

9.3 Application 9-2: A Subprogram to Format Your Output

In Chapter Three we discussed the use of the **INT** function to round numbers. Since you have used this technique in many programs, you have probably discovered that the technique had some shortcomings. While it did round the numbers correctly, it did not cause the output to be lined up in neat columns with all of the decimal points in a line. It also had no effect on numbers printed in exponential notation.

In this section we will develop a program which will not only round the numbers to two decimal places but will also see that they are right-justified, or printed in the rightmost columns of a fixed number of columns. In this way you can make certain that all of your numbers are lined up.

First we must decide on some conventions. Suppose we choose to use a set of 10 columns for our numbers. If we have a number which is not exponential, it will be printed as

```
xxxxxxx.xx
```

where some of the places in front of the decimal point might be blank.

What should we do about exponential numbers? Let's adopt the rule that numbers of this type will be right justified in the 10 columns and will display 2 figures after the decimal point. That means that exponential numbers will always look like

```
--x.xxE+xx
```

where the dashes indicate a blank space. This is an arbitrary decision, and you might prefer to display such numbers differently. Once you understand the program below, you can modify it to suit your own preferences.

Here is the solution procedure.

```
initialize: get number of columns (C)
       get the number to be formatted

test:  test if number is exponential
        convert number to a string
        pull off the rightmost four characters
        test if the ASCII value is 69 (This is the ASCII
             code for E and indicates an exponential
             number. Remember that using the ASC function
             only looks at the ASCII code for the first
             character of the string.)
        if yes, goto exponential format
        if no, goto normal format

normal format: multiply number by 100, round and truncate
       convert to string
       get length of string
       if length is greater than C-1, then fill with
          asterisks (in this case, number is too big for
          columns)
       if length is less than 2, then add enough zeroes to
          bring length to 2
      if length is less than C-1, add blanks in front
      add decimal point
      exit

exponential format: convert number to string
       divide string into 4 rightmost characters,
          remaining
          left characters
       convert left string into number
       format this in C-4 columns (remaining 4 for
          exponent)
          goto normal format
      add exponent
      exit
```

Here is the program. When you understand the logic, you will probably want to store this as a text file so that you can EXEC it as a subroutine in future programs.

```
10 C1=10
20  INPUT X1
200 S$="           ": REM 10 BLANK SPACES
210 Y$=STR$ (X1):R$=RIGHT$ (Y$,4)
220  IF ASC (R$) < > 69 THEN X=X1:C=C1: GOSUB 500:
      GOTO 250
230 C=C1 - 4: GOSUB 600: GOTO 250
250  PRINT Y$
260  END
500  REM NORMAL FORMAT
510 Y=INT (X * 10 ^ 2 + .5)
520 Y$=STR$ (Y):L=LEN (Y$)
```

```
530  IF L > C - 1 THEN Y$="": FOR I=1 TO C:Y$=Y$ + "*":
       NEXT I: GOTO 560
540  IF L < C - 1 THEN Y$=LEFT$ (S$,C - L - 1) + Y$
550 Y$=LEFT$ (Y$,C - 3) + "." + RIGHT$ (Y$,2)
560  RETURN
600  REM EXPONENTIAL FORMAT
610 L=LEN (Y$):L$=LEFT$ (Y$,L - 4)
620 X=VAL (L$): GOSUB 500
630 Y$=Y$ + R$: RETURN
```

9.4 What is Word Processing?

Microcomputers are currently causing an office revolution. As microcomputers become cheaper and easier to use, they are finding their way into every aspect of business. Nowhere does the revolutionary impact of microcomputers promise to be greater than in the area of word processing.

In brief, a **word processor** is a device made by combining the traditional typewriter with the capabilities of the computer for storing, editing, retrieving, displaying, and printing information. It is no exaggeration to say that the traditional typewriter is now as obsolete as a Model T. Over the next decade or so the typewriter will be completely replaced by increasingly sophisticated word processors.

The basic concept of a word processor is to use the microcomputer as a typewriter. Instead of using paper to record the words, you use the computer memory. First, the words are stored in RAM. When you wish to make a permanent record of them, you store them on disk as a data file. As you type, the text can be viewed on the video display. This part of word processing is not revolutionary.

The true power of a word processor doesn't come into play until you need to edit the data in a document. By using the power of the computer, you can perform the following tasks quickly and with little effort: move to any point in the document; add words, phrases, sentences, or even paragraphs; delete portions of the text; move a block of text from one part of the document to another; insert "boiler-plate" information (standard pieces of text such as resumes or company descriptions) from another data file (for example, you could add a name and address from a mailing list); selectively change all occurrences of one word (say, "John") to another (say, "Jim"); or print the contents of a file according to a requested format.

All of the above operations are possible because the computer is able to manipulate strings in addition to numbers. In fact, you may turn your Apple II Computer into quite a respectable word processor. We will discuss some available packages in Chapter Twelve. In Section 9.6, to give you a taste of actual word processing, we will build a rudimentary word processor which you can use to prepare letters, term papers, memos, or other documents.

Using Your Printer

For actual word processing you will probably want to have a printer available. Recall what we have covered up to this point concerning the printer. The printer is controlled by an interface board placed in one of the slots of the Apple II Computer. These interface boards can be either serial or parallel cards depending upon your printer and the way the data is sent from the computer to the printer. For clarity let's assume that the card controlling the printer is placed in slot 1. The instruction

```
10 PR#1
```

will then direct all further **PRINT** statements to the printer in addition to the screen. To go back to displaying the results of **PRINT** statements only on the screen, we use the instruction

```
20 PR#0
```

The printer is built to accept a stream of ASCII character codes. Some of these correspond to letters and symbols to be printed, and some correspond to control characters which make the printer perform various non-print functions (such as carriage return, line feed, and space to the top of the next page). To the printer a line consists of a sequence of printable characters followed by a **RETURN**. The **RETURN** actually tells the printer two things. First, it causes a carriage return; second, it causes the paper to advance one line (called a **line feed**).

A typical printer has controls for setting a number of different parameters. Among these switches are those which control line width, margins, number of lines per inch, number of lines per page, and size of the characters. You must refer to your printer manual for the correct procedures for setting these switches because they are not the same for all printers. (In the case of some printers, these various settings can be controlled by sequences of characters from the computer. These so-called **ESCAPE sequences** are displayed as if they were actual data. However, the printer recognizes them as commands and does not print them.)

Many printers will not honor the TAB command, so you will have to use SPC instead. Also, to print a blank line you may have to use PRINT " ". Finally, the HOME command has no meaning to the printer, but you may have the ability to go to the top of a new page if your printer allows form feed control through a program.

9.5 Form Letters

You may use the string-manipulating capability of your computer to prepare form letters that look like genuine correspondence. Let us illustrate the technique by preparing the following form letter.

April 1, 1984

Dear Mr./Ms.:

All of us at Neighborhood Building Supplies, Inc. have appreciated your patronage in the past. We are writing to let you know we will be moving our store to 110 S. Main St., effective July 1.

You will find the new store larger and more convenient than the old one. In addition, we will now stock a more extensive line of energy-efficient doors and windows.

We look forward to your continued patronage. Please let us know if we may be of assistance in your building plans.

Cordially yours,

Samuel Gordon,
President
Neighborhood Building Supplies, Inc.

Suppose this letter is to go out to 100 customers on a mailing list. Also assume the mailing list is maintained in a data file on diskette.

Associated with each customer are six variables: a numeric variable which equals 0 if the customer is male and 1 if female and an address which is stored as five strings corresponding to the name of the individual, company name, street address, city, and state zip code. We need to store city separately from state and zip code because we usually have a comma separating city and state. Because the comma serves as a delimiter to separate one string from another, the computer would never finish reading the state name. Finally, suppose that the name of the file is CUSTOMER. Write a program to produce the desired stack of 100 letters.

Our program will consist of two parts. The first will allow us to type in the body of the letter. The various lines of the letter will be typed exactly as if we were using a typewriter. The computer will use a string array A$(J) to store the body of the letter, with A$(1) holding the first line, A$(2) the second, and so forth. We will indicate to the computer that the body of the text is complete by typing % followed by **RETURN**.

As soon as the character % is recognized, the program goes into its second phase: the actual generation of the letters. The program opens the address file for output. One by one, it reads the address entries. After reading a given entry, it prints the date at the top of the letter, followed by the address. Next, the program determines the last name of the addressee from the first line of the address and inserts it after the "Dear." Finally, the body of the letter is printed.

Here is the solution structure.

```
body:  type in body of letter
       count number of lines
loop for each customer:
        print date
```

```
print blank line
print address lines
find last name:
    search from right of string looking for first blank.
When blank is found, pull off rightmost characters as last
name.
check if male or female and append MR. or MS.
write greeting and blank line
write body of letter
endloop
```

Here is the program which accomplishes all of this.

```
10 DIM A$(100),B$(6,100),B(100)
20 PRINT "AFTER EACH ? TYPE ONE LINE OF
   THE LETTER,"
30 PRINT "FOLLOWED BY RETURN"
40 PRINT "TO END LETTER, TYPE % FOLLOWED
   BY RETURN"
50 J=1 : REM J IS THE NUMBER OF LINES
60 INPUT A$(J)
70 IF A$(J)="%" THEN 100
80 J=J+1: REM NEXT LINE
90 GOTO 60
100 PRINT CHR$(4); "OPEN CUSTOMER"
110 PRINT CHR$(4); "READ CUSTOMER"
115 FOR N=1 TO 100: REM N=CUSTOMER #
120 INPUT B(N), B$(1,N), B$(2,N), B$(3,N), B$(4,N),
    B$(5,N)
130 NEXT N :REM GO TO NEXT CUSTOMER
140 PRINT CHR$(4); "CLOSE CUSTOMER"
150 PR#1: REM ADDRESS PRINTER
155 FOR N=1 TO 100
160 PRINT A$(1): REM PRINT DATE
170 PRINT " ": REM PRINT BLANK LINE
175 REM 150-180 PRINT ADDRESS
180 PRINT B$(1,N)
181 PRINT B$(2,N)
182 PRINT B$(3,N)
183 PRINT B$(4,N)","B$(5,N): PRINT " "
190 REM FIND LAST NAME OF ADDRESSEE
200 L=LEN(B$(1,N))
210 FOR K=L-1 TO 1 STEP -1
220 C$=MID$(B$(1,N),K,1)
230 IF C$=" " THEN 300
235 GOTO 240: REM TEST FOR SPACE BETWEEN WORDS
240 NEXT K
300 L$=RIGHT$(B$(1,N),L-K): REM L$=LAST NAME
305 S$="MR. ":IF B(N)=1 THEN S$="MS. "
310 PRINT "DEAR ";S$;L$;":": PRINT  " "
320 FOR M=2 TO J :REM PRINT BODY OF LETTER
330 PRINT A$(M)
```

```
340 NEXT M: REM GO TO NEXT LINE OF LETTER
350 NEXT N : REM GO TO NEXT CUSTOMER
360 PR#0: REM TURN SCREEN ON
400 END
```

For the above program to work properly, you should type in the date as the first line of the letter. You should not type in the line which begins "Dear" because the program generates this line for you.

If you have an Apple IIe, you will be able to use both capital and lower case letters.

If your printer allows control of form feed, you will want to add this at line 345 so each letter is on a separate sheet. If not, you will want to add enough blank lines to accomplish this. To do this we have:

J lines in the letter + 4 address lines + 1 greeting line + 3 blanks = 8 + J lines.

On a typical sheet there are 66 lines, so you will have to print out 66 − (8 + J) blank lines after the body of the letter. You can accomplish this with a FOR...NEXT loop.

The above program was used in the context of a specific letter. Please note that the program may be used to generate any set of form letters from an address list. In the exercises we will suggest some modifications which you can use to generate invoices or other correspondence with variable text in the body of the letter.

Exercises

1. Add a second page to the Neighborhood Building Supplies form letter. At the top of the page should be a date and the page number. The title should be "OPENING SPECIAL". The text should consist of the following message:

   This letter is being sent only to our most valued customers! Bring this coupon with you for a 10 percent discount on any order placed in the month of JULY, 1984.

2. Change the Neighborhood Building Supplies form letter so that in the third and fourth sentences, the name of the addressee is used (for example, "Ms. Thomas, you will find . . .").
3. Write a program that prints invoices. Assume that the invoices are stored in a file called "INVOICE", where a particular invoice contains for each item shipped: quantity, price, item description (limited to 15 characters), and total cost. Assume the invoice entry starts with a four-string entry giving the customer name, address, and date. The file entry for a given invoice ends with the character %. You may assume the first entry in the file is the number of invoices contained in the file. Your program should print out invoices corresponding to all entries in the file.

4. Suppose the file INVOICE of Exercise 3 contains only a customer identification number rather than a customer name and address. Suppose the whole list of customer addresses contains customer identification numbers. Modify the program of Exercise 3 so it locates the customer name and address automatically.
5. Assume a change in local ordinances now allows you to not charge local sales tax to any customer who lives outside the city limits. Suppose that city consists of ZIP CODE 91723. Modify the program of the text so that it checks the ZIP CODE of the customer. For customers not in ZIP CODE 91723, insert the following paragraph in the letter:

   Good news! You will no longer be charged local sales tax, in accordance with the change in local ordinances. This will yield even further savings from our already low prices.

9.6 Application 9-3: A Do-It-Yourself Word Processor

It is really quite impractical for you to build your own word processor. For one thing, such a program is long and complicated. Moreover, if you write in BASIC, the operation of the program will tend to be rather slow. An efficient word processor will almost always be written in machine language. Nevertheless, in order to acquaint you with a few of the virtues of word processing, let's ignore what I just said and build a word processor anyway!

Our word processor will be line oriented. You will type in each line just as if you are using a typewriter. At the end of each line you will cause a carriage return by hitting the RETURN key. The Jth line will be stored in the string variable A$(J).

Our word processor will have five modes. In the first mode we input the text of our document. This operation will proceed exactly as if you were using a typewriter. At the beginning of each line, the word processor will display a ?. You type your line after the question mark. Terminate the line by hitting the **RETURN** key. In order to indicate that you do not wish to type any more lines, type % followed by **RETURN**.

A second mode allows us to save our document. For the purposes of this word processor, let's assume you have a diskette file. The program then saves your document as a data file under a file name requested by the program. The first item in a document file will always be the number of lines in the document. This quantity will be denoted by the variable L. Next come the lines of the document: A$(1), A$(2), . . . , A$(L) .

A third mode lets you produce a draft version of the document. In this mode the document is printed with each line preceded by its line number. The line numbers allow you to easily identify lines having errors. In order to print a document, you must first save it on the disk.

A fourth mode allows document editing. To correct errors you identify the line by number and retype the line. To end the edit session type % followed by **RETURN**. This will bring you back to the beginning of the program, but

you will still be working on the same document. After ending an edit session, your customary next action should be to save the document. The fifth and final mode allows printing of a final draft of a document.

When the word processor is first run, you will see the following prompt:

```
WORD PROCESSING PROGRAM
CHOOSE ONE OF THE FOLLOWING MODES
     INPUT TEXT(I)
     DISPLAY DRAFT (DD)
     PRINT FINAL DRAFT (PF)
     SAVE FILE (S)
     EDIT (E)
     QUIT (Q)
```

In response, you type I, DD, PF, S, E, or Q followed by RETURN. If you choose I, the screen will be cleared and you may begin typing your document. For the other modes, there are prompts to tell you what to do.

Here is a listing of the program.

```
5 DIM A$(150)
10 PRINT "WORD PROCESSING PROGRAM"
20 PRINT "CHOOSE ONE OF THE FOLLOWING MODES"
30 PRINT , "INPUT TEXT(I)"
40 PRINT , "DISPAY DRAFT(DD)"
50 PRINT , "PRINT FINAL DRAFT(PF)"
60 PRINT , "SAVE FILE(S)"
70 PRINT , "EDIT(E)"
80 PRINT , "QUIT(Q)"
90 INPUT X$
100 IF X$="I" THEN 1000
110 IF X$="DD" THEN 2000
120 IF X$="PF" THEN 3000
130 IF X$="S" THEN 4000
140 IF X$="E" THEN 5000
150 IF X$="Q" THEN 6000
160 GOTO 90: REM IF X$ DOES NOT MATCH ANY
    OF THE PROMPTS
1000 L=1
1010 INPUT A$(L)
1020 IF A$(L)="%" THEN 10
1030 L=L+1
1035 IF L<=150 THEN 1010
1040 IF L>150 THEN PRINT "DOCUMENT TOO
     LARGE"
1050 GOTO 10
2000 INPUT "DOCUMENT NAME";Y$
2010 PRINT CHR$(4); "OPEN "; Y$
2015 PRINT CHR$(4); "READ "; Y$
2020 INPUT L
2030 FOR K=1 TO L
2040 INPUT A$(K)
2050 PRINT K;">";A$(K)
2060 NEXT K
```

```
2070 PRINT CHR$(4); "CLOSE "; Y$
2090 GOTO 10
3000 INPUT "DOCUMENT NAME";Y$
3010 PRINT CHR$(4); "OPEN "; Y$
3015 PRINT CHR$(4); "READ "; Y$
3020 INPUT L
3030 FOR K=1 TO L
3040 INPUT A$(K)
3050 NEXT K
3060 PRINT CHR$(4) "CLOSE "; Y$
3070 PR#1
3080 FOR K=1 TO L
3090 PRINT A$(K)
3100 NEXT K
3110 PR#0
3120 GOTO 10
4000 INPUT "DOCUMENT NAME";Y$
4010 PRINT CHR$(4); "OPEN "; Y$
4015 PRINT CHR(4); "WRITE "; Y$
4020 PRINT L
4030 FOR K=1 TO L
4040 PRINT A$(K)
4050 NEXT K
4060 PRINT CHR$(4); "CLOSE "; Y$
4070 GOTO 10
5000 INPUT "DOCUMENT NAME"; Y$
5010 PRINT CHR$(4); "OPEN ";Y$
5015 PRINT CHR$(4); "READ "; Y$
5020 INPUT L
5030 FOR K=1 TO L
5040 INPUT A$(K)
5050 NEXT K
5055 PRINT CHR$(4); "CLOSE "; Y$
5060 INPUT "NUMBER OF LINE TO EDIT";Z
5070 HOME
5080 PRINT A$(Z)
5090 INPUT "TYPE CORRECTED LINE";A$(Z)
5100 IF A$(Z) <> "%" THEN 5060
5110 GOTO 10
6000 END
```

Use this program to type a few letters. You will find it a big improvement over a conventional typewriter. This will probably whet your appetite for the more advanced word processing features we described in the preceding section.

Exercises

1. Modify the word processor to allow input of line width. (You will not be able to display lines longer than 40 characters on a single line. Remember string variables may contain up to 255 characters.)

2. Modify the word processor so that you may extend a line. This modification should let your corrected line spill over into the next line of text. The program should then correct all subsequent lines to reflect the addition.
3. Modify the word processor to allow deletions from lines. Subsequent lines should be modified to reflect the deletion.

10
Playing with Pictures

In many applications it is helpful to present data in pictorial form. By displaying numerical information in graphic form, it is often possible to develop insights and draw conclusions which are not immediately evident from the original data.

In this chapter we will discuss procedures for using your Apple II Computer to create various kinds of pictorial displays on the screen. Such procedures belong to the field of **computer graphics**. This chapter will introduce you to this field.

10.1 Low Resolution Graphics

There are three display modes in the Apple II: **text mode, low resolution graphics mode**, and **high resolution graphics mode**. Up to this point you have been using the text mode to display characters on the screen. The low resolution graphics mode lets you draw figures in as many as 16 colors. High resolution allows you to draw more detailed figures, but only in eight colors.

In order to obtain color, you must hook your Apple II Computer up to a color monitor (a color television set will do). If you use a black and white monitor, you may use the graphics modes exactly as described below, but you will generate only black and white pictures.

You may select among the three modes by using the following Applesoft BASIC commands.

TEXT = text mode
GR = low resolution graphics mode
HGR = high resolution graphics mode

When BASIC is started, the display is automatically in text mode. The above commands may be used to switch from one display mode to another, either within a program or via a keyboard command. If you are entering the TEXT command via keyboard on the Apple IIe, you can simply hit the ESCape key followed by T.

Text Mode

As we have already noted, the video display of the Apple II Computer is capable of displaying 24 rows of 40 characters each. This gives us up to

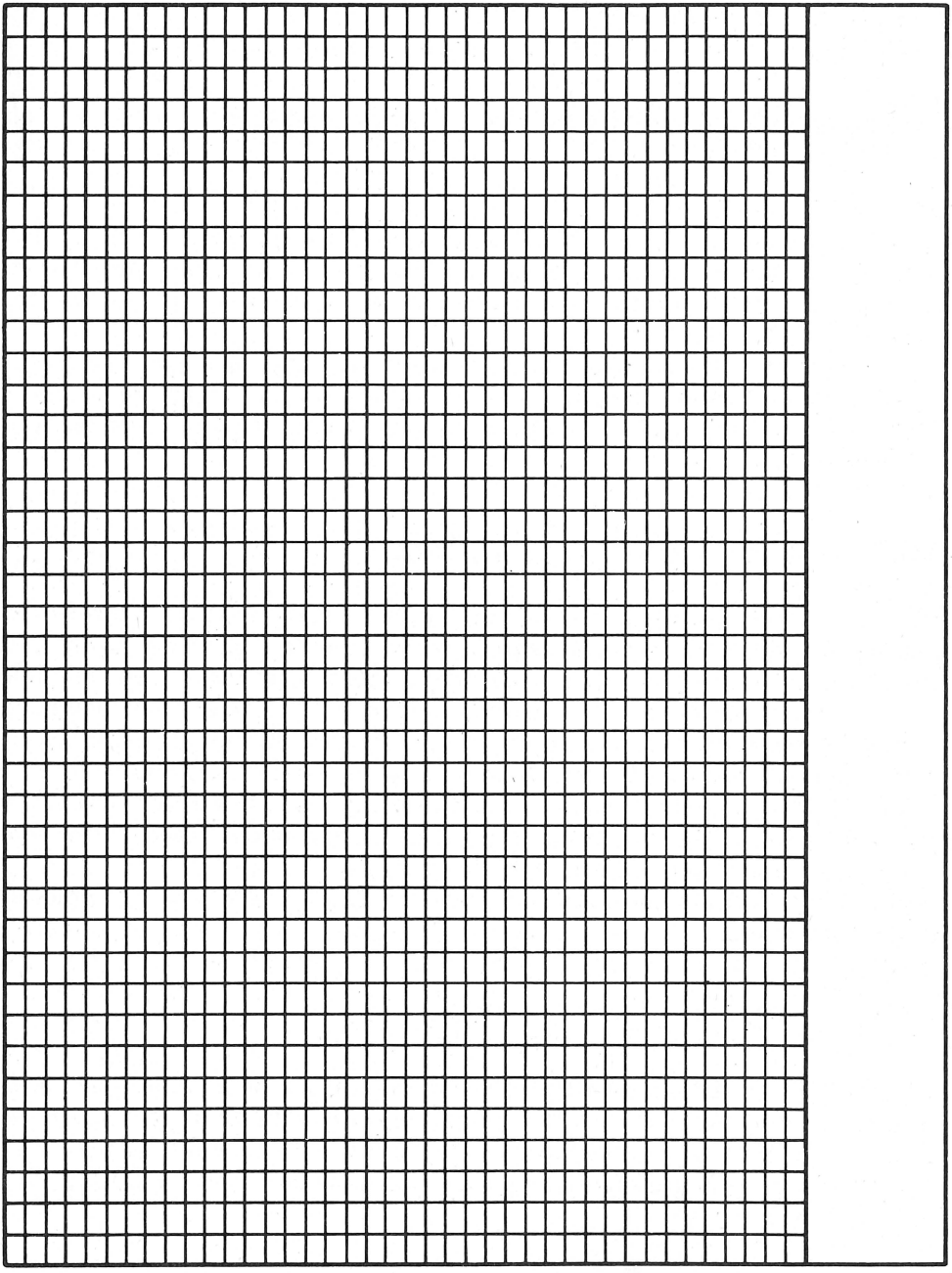


Figure 10-1. Screen layout for low resolution graphics mode.

24 × 40 or 960 possible character positions. These various character positions divide the screen into small rectangles. One rectangle corresponds to each character position.

Low Resolution Graphics Mode

In low resolution graphics mode, the screen is divided into the same number of columns as for text mode, but each text row is divided into two low resolution graphics rows. We have indicated in Figure 10-1 the subdivision of the screen corresponding to the text mode.

The rectangles we have divided the screen into are arranged in rows and columns. We number the rows from 0 to 47, with row 0 being at the top of the screen and row 47 at the bottom. The columns are numbered from 0 to 39, with column 0 being at the extreme left and column 39 at the extreme right. Each rectangle on the screen is identified by a pair of numbers, indicating the row and column with the column number first. For example, we have indicated the rectangle in the twelfth row and sixteenth column in Figure 10-2.

The low resolution graphics command **GR** blocks out the top 20 lines of the screen, giving a 40 x 40 grid of rectangles for plotting. The bottom 4 lines may be used for text. The plotting color is automatically set to black by the **GR** command. This means that you won't be able to see anything you plot since it will appear as black on black.

The first chore on entering low resolution graphics mode is to choose the color. Sixteen colors are possible, numbered as follows:

0—black	8—brown
1—magenta	9—orange
2—dark blue	10—grey
3—purple	11—pink
4—dark green	12—green
5—grey	13—yellow
6—medium blue	14—aqua
7—light blue	15—white

To set the color we use an instruction of the form

```
COLOR=11
```

Once the color has been set, all subsequent plotting will be in that color until you give another **COLOR** instruction.

To light up the rectangle at position (x,y) in the current color, we use the graphics instruction

```
10 PLOT X,Y
```

Here x denotes the column number and y denotes the row number. To turn off the rectangle, we plot it in color 0 (black).

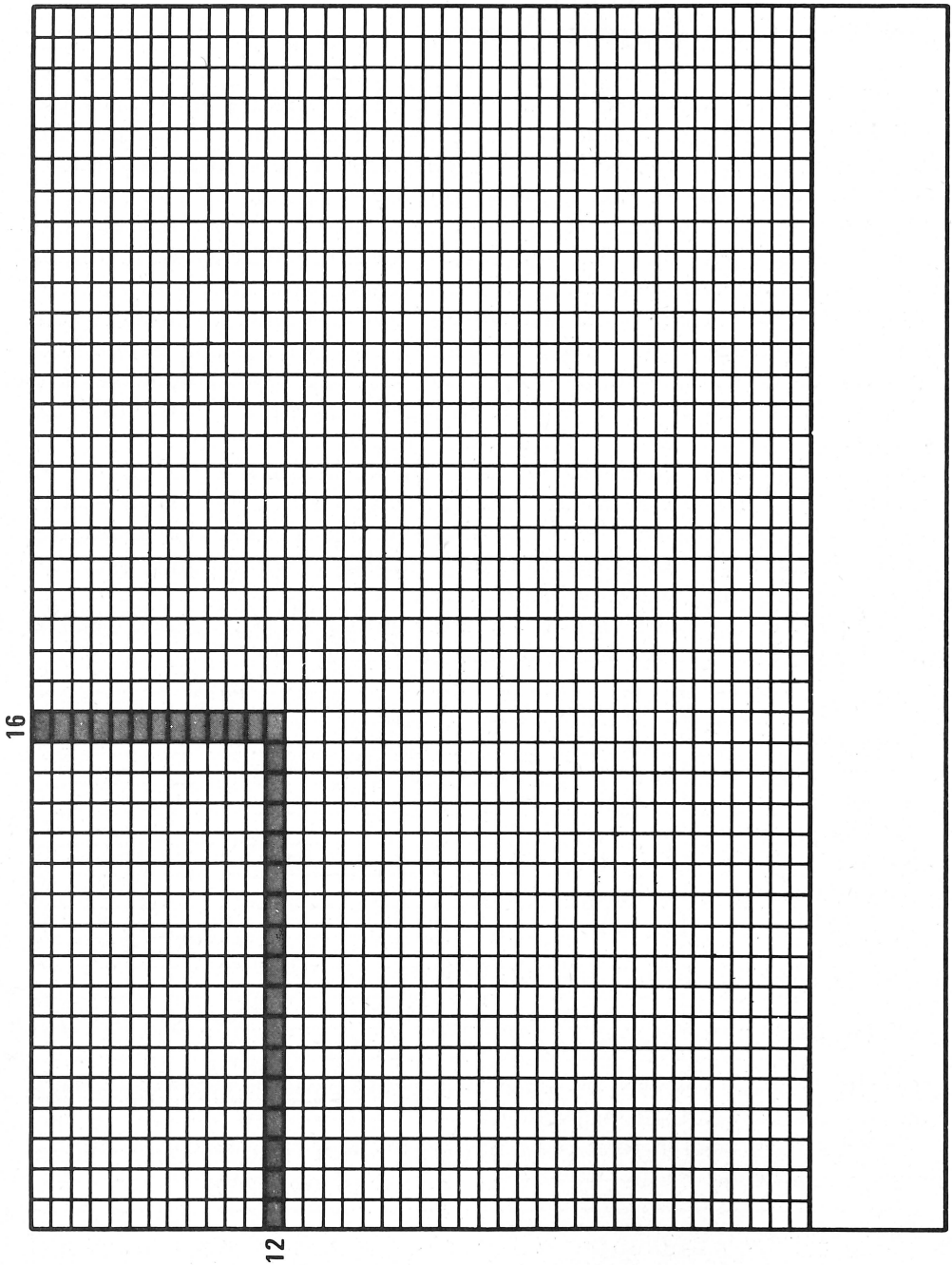


Figure 10-2.

TEST YOUR UNDERSTANDING 1 (answer on page 245)

Write a set of instructions to illuminate the graphics rectangle at row 18, column 22, in color 12.

Low resolution graphics has special commands for drawing horizontal and vertical lines. To draw a horizontal line from column 2, row 11 to column 20, row 11, we use an instruction of the form

```
20 HLIN 2,20 AT 11
```

Similarly, to draw a vertical line from column 5, row 1 to column 5, row 20, we use an instruction of the form

```
30 VLIN 1,20 AT 5
```

Example 1. Write a program which draws a dark green horizontal line across row 10 of the screen.

Solution.

```
10 GR
20 COLOR=4
30 HLIN 0,39 AT 10
40 END
```

Example 2. Write a program which draws a dark green vertical line in column 25 from row 5 to row 15. The program should blink the line 50 times.

Solution. The blinking effect may be achieved by repeatedly clearing the screen using GR. Here is our program.

```
10 FOR K=1 TO 50:REM K CONTROLS BLINKING
20 GR
30 COLOR=4
40 VLIN 5,15 AT 25
50 NEXT K
60 END
```

TEST YOUR UNDERSTANDING 2 (answer on page 245)

Write a set of instructions to blink the graphics rectangle at row 18, column 22, in color 12.

TEST YOUR UNDERSTANDING 3 (answer on page 245)

Write a program to draw a vertical line from row 2 to row 20 in column 8.

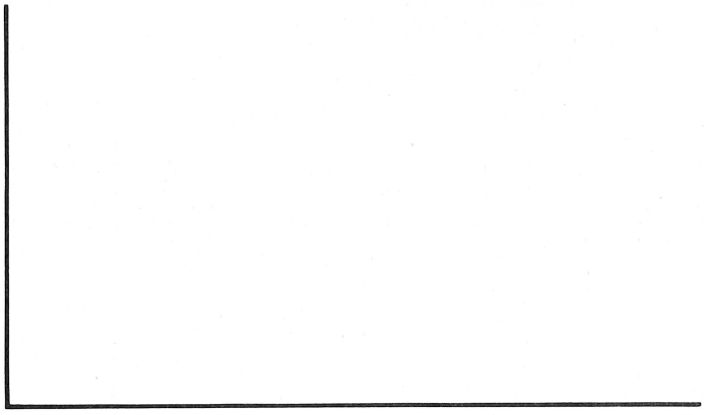


Figure 10-3.

Example 3. Draw a pair of x and y axes as shown in Figure 10-3. Draw the axes in light blue.

Solution.

```
10 GR
20 COLOR=7
30 HLIN 0,39 AT 39
40 VLIN 0,39 AT 0
50 END
```

Your program may determine the color of a given rectangle on the screen using the **SCRN** instruction. For example, **SCRN(7,3)** is equal to the color number of rectangle 7,3.

Exercises (answers on page 321)

Draw the following straight lines.

1. A horizontal line completely across the screen in row 18.
2. A vertical line completely up and down the screen in column 17.
3. A pair of straight lines which divide the screen into four equal squares.
4. Horizontal and vertical lines which convert the screen into a tic tac toe board.
5. A vertical line of double thickness from rows 1 to 24 in column 30.
6. A diagonal line going through the character positions (1,1), (2,2), . . ., (24,24).
7. A horizontal line with "tick marks" as follows:

8. A vertical line with "tick marks" as follows:

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, and 3

1:
```
10 GR
20 COLOR=12
30 PLOT 22,18
40 END
```
2:
```
10 GR
20 COLOR=12
30 PLOT 22,18
40 GOTO 10
60 END
```
3:
```
10 GR
20 COLOR=1
30 VLIN 2,20 AT 8
40 END
```

Computer Art

The computer may be used to create interesting pieces of random graphics art. We use low resolution graphics and the random number generator function (RND) to give the computer free "artistic license" to create random patterns on the screen. Let us create a program which runs down the rows of graphics blocks and randomly illuminates some of them. We will control the number of blocks to be illuminated by a "density factor" which we input. This factor will be a number between 0 and 1 and will be denoted by the variable D (for density).

Our program will consider each graphics block separately. It will use the output of RND to make the decision on whether to illuminate the block or not. If RND is less than D, the block will be illuminated; if RND is at least D then the block will not be illuminated. Here is our program.

```
 5 INPUT "DENSITY FACTOR"; D
 6 GR
10 FOR R=0 TO 39 : REM R=ROW NUMBER
20 FOR C=0 TO 39: REM C=COLUMN NUMBER
30 COLOR=INT(16*RND(1))
40 IF RND(1) < D THEN 60
50 GOTO 100
60 PLOT C,R
```

```
100 NEXT C
110 NEXT R
200 END
```

Run this program for assorted values of D. Some suggested values are D = .1, .5, and .7, respectively. Because of the unpredictability of the random number generator, the same value of D will usually yield quite different pictures on successive runs.

Exercises

1. Run the above program three times for the value D = .4.
2. Run the above program for the values D = .1, .2, .3, . . . , 1.0. Can you predict the display for D = 1.0 in advance?

10.2 Application 10-1: Tic Tac Toe

In this section we present a program for the traditional game of tic tac toe. We won't attempt to let the computer execute a strategy: we will let it be fairly stupid and choose its moves randomly. We will also use the random number generator to "flip" for the first move. Throughout the program you will be "O" and the computer will be "X." Figures 10-4 and 10-5 show the output from a sample game.

TEST YOUR UNDERSTANDING 1 (answer on page 252)

How can the computer toss to see who goes first?

The computer draws a tic tac toe board. See Figure 10-6. The computer displays your move and makes a move of its own. See Figure 10-7.

The computer makes its move and so on until someone wins or a tie game results.

The program TIC TAC TOE makes use of much of what we have learned. First of all, the program uses the graphics mode to draw the tic tac toe board, beginning in line 2000. We have structured the program so that it consists of a series of subroutines. Lines 2000-2999 contain the instructions to draw the board and to display the current status of the game. Lines 3000-3999 contain a subroutine which inputs your move. Lines 4000-4999 contain a subroutine which lets the computer decide its move. Lines 5000-5999 process the current move and decide if the game is over. The logic in lines 6000-8000 allows the computer to determine if there is a winning move or if it should block its opponent.

Here are the variables used in the program.

Z = 0 if it's your move and Z = 1 if it is the computer's
A$(J) (J = 1,2, . . . ,9) contain either O, X, or the empty string, indicating the current status of position J

S = the position of the current move
M = the number of moves played (including the current one)

A video display worksheet was used to lay out the board and to determine the coordinates for the lines and the X's and O's.

Here is a listing of our program.

```
10  DIM A$(9)
20  DIM B$(3,3,3)
30  HOME
40  PRINT "TIC TAC TOE"
50  PRINT "YOU WILL BE O; THE COMPUTER WILL BE X"
60  PRINT "THE POSITIONS ON THE BOARD ARE NUMBERED"
70  PRINT "AS FOLLOWS"
80  PRINT "1,2,3"
90  PRINT "4,5,6"
100  PRINT "7,8,9"
110  PRINT "THE COMPUTER WILL TOSS FOR FIRST"
120  IF RND (1) > .5 THEN 130
125  GOTO 160
130  PRINT "YOU GO FIRST"
140  LET Z=0
150  GOTO 180
160  PRINT "I'LL GO FIRST"
170  LET Z=1
180  PRINT "WHEN READY TO BEGIN TYPE 'R'"
190  INPUT C$
200  IF C$="R" THEN 210
205  GOTO 180
210  GOSUB 2000
300  FOR M=1 TO 9: REM M=MOVE#
```

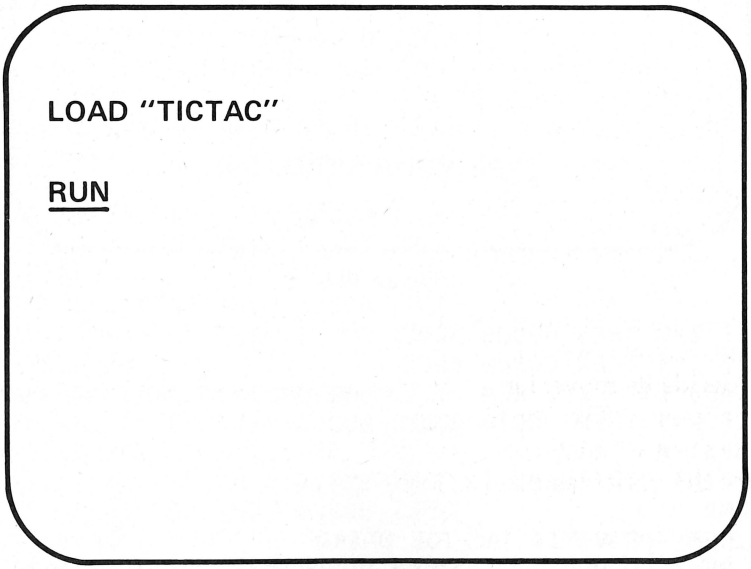


Figure 10-4.

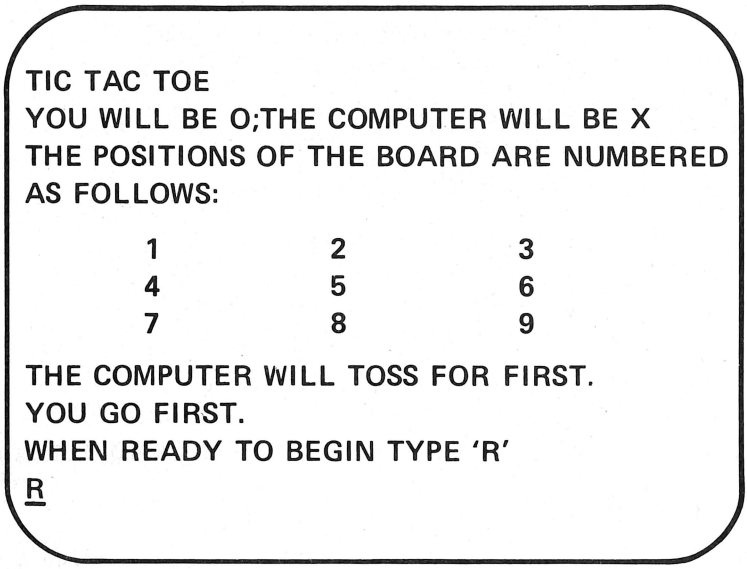

TIC TAC TOE
YOU WILL BE O;THE COMPUTER WILL BE X
THE POSITIONS OF THE BOARD ARE NUMBERED
AS FOLLOWS:
1 2 3
4 5 6
7 8 9
THE COMPUTER WILL TOSS FOR FIRST.
YOU GO FIRST.
WHEN READY TO BEGIN TYPE 'R'
R

Figure 10-5.

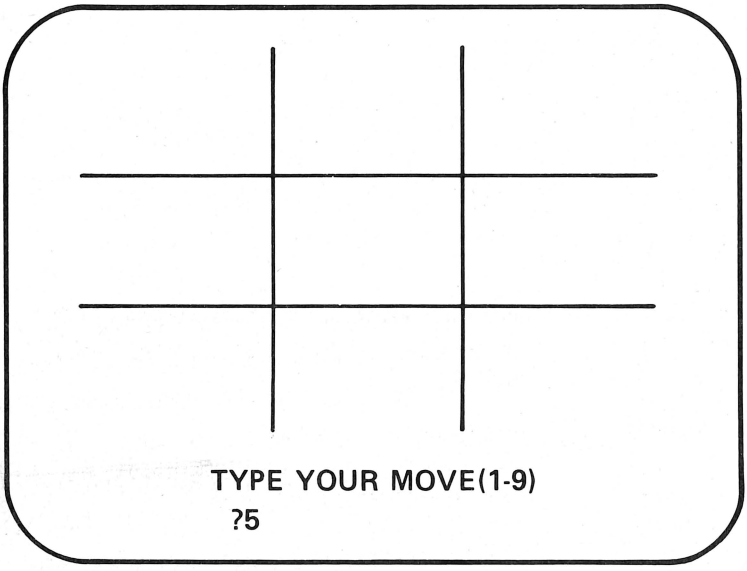


Figure 10-6.

```
310  IF Z=0 THEN GOSUB 3000
320  IF Z=1 THEN GOSUB 6000
330 Z=ABS (1 - Z): REM    Z=1 MEANS NEXT MOVE IS COMPUTER
350  IF W=1 THEN  GOTO 400
355  NEXT M
360  PRINT "THE GAME IS TIED"
400  END
2000  REM  DRAW TIC TAC TOE BOARD
2010  GR : COLOR=1
2020  FOR J=1 TO 2
```

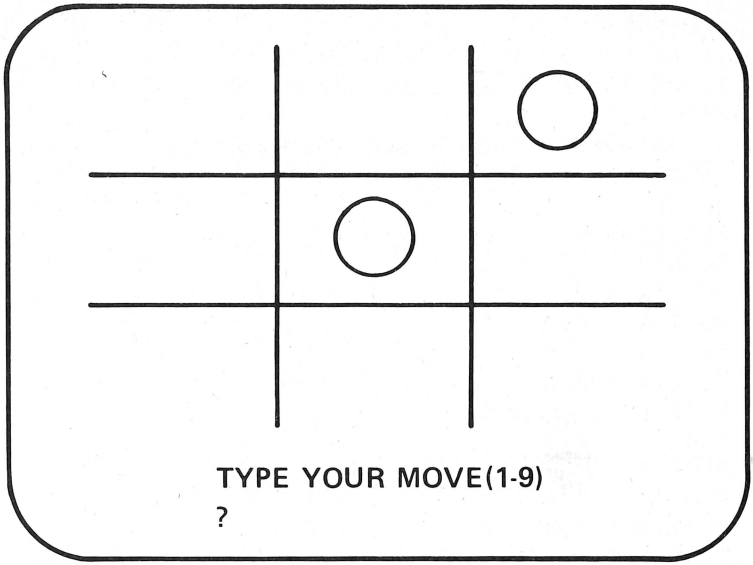


Figure 10-7.

```
2030  FOR K=0 TO 39
2040  PLOT K,13 * J
2050  NEXT K
2060  NEXT J
2070  FOR J=1 TO 2
2080  FOR K=0 TO 39
2090  PLOT 13 * J,K
2100  NEXT K
2110  NEXT J
2115  REM   DISPLAY CURRENT GAME STATUS
2120 A=7:B=7
2121  IF A$(1)="X" THEN GOSUB 2300
2122  IF A$(1)="O" THEN GOSUB 2200
2123 A=20:B=7
2124  IF A$(2)="X" THEN GOSUB 2300
2125  IF A$(2)="O" THEN GOSUB 2200
2126 A=33:B=7
2127  IF A$(3)="X" THEN GOSUB 2300
2128  IF A$(3)="O" THEN GOSUB 2200
2129 A=7:B=20
2130  IF A$(4)="X" THEN GOSUB 2300
2131  IF A$(4)="O" THEN GOSUB 2200
2132 A=20:B=20
2133  IF A$(5)="X" THEN GOSUB 2300
2134  IF A$(5)="O" THEN GOSUB 2200
2135 A=33:B=20
2136  IF A$(6)="X" THEN GOSUB 2300
2137  IF A$(6)="O" THEN GOSUB 2200
2138 A=7:B=33
2139  IF A$(7)="X" THEN GOSUB 2300
2140  IF A$(7)="O" THEN GOSUB 2200
```

```
2141 A=20:B=33
2142  IF A$(8)="X" THEN GOSUB 2300
2143  IF A$(8)="O" THEN GOSUB 2200
2144 A=33:B=33
2145  IF A$(9)="X" THEN GOSUB 2300
2146  IF A$(9)="O" THEN GOSUB 2200
2150  RETURN
2200  REM DRAW O
2205  COLOR=1
2210  PLOT A - 2,B
2211  PLOT A - 2,B + 1
2212  PLOT A - 2,B - 1
2213  PLOT A + 2,B
2214  PLOT A + 2,B + 1
2215  PLOT A + 2,B - 1
2216  PLOT A,B - 2
2217  PLOT A - 1,B - 2
2218  PLOT A + 1,B - 2
2219  PLOT A,B + 2
2220  PLOT A - 1,B + 2
2221  PLOT A + 1,B + 2
2230  RETURN
2300  REM DRAW X
2305  COLOR=1
2310  PLOT A,B
2311  PLOT A - 1,B - 1
2312  PLOT A - 1,B + 1
2313  PLOT A - 2,B - 2
2314  PLOT A - 2,B + 2
2315  PLOT A + 1,B + 1
2316  PLOT A + 1,B - 1
2317  PLOT A + 2,B + 2
2318  PLOT A + 2,B - 2
2320  RETURN
3000  PRINT "TYPE YOUR MOVE"
3010  INPUT S
3020 A$(S)="O"
3030  GOSUB 2000
3040  GOSUB 5000
3050  RETURN
4000  LET S=INT (9 * RND (1) + 1)
4010  IF A$(S)="" THEN 4060
4015  GOTO 4000
4060 A$(S)="X"
4070  GOSUB 2000
4080  GOSUB 5000
4090  RETURN
5000  IF Z=0 THEN C$="O"
5005  IF Z=1 THEN C$="X"
5010  IF A$(1)=A$(2) THEN 5012
5011  GOTO 5020
5012  IF A$(1)=A$(3) THEN 5014
```

```
5013  GOTO 5020
5014  IF A$(3)=C$ THEN 5900
5020  IF A$(1)=A$(4) THEN 5022
5021  GOTO 5030
5022  IF A$(4)=A$(7) THEN 5024
5023  GOTO 5030
5024  IF A$(7)=C$ THEN 5900
5030  IF A$(1)=A$(5) THEN 5032
5031  GOTO 5040
5032  IF A$(5)=A$(9) THEN 5034
5033  GOTO 5040
5034  IF A$(9)=C$ THEN 5900
5040  IF A$(2)=A$(5) THEN 5042
5041  GOTO 5050
5042  IF A$(5)=A$(8) THEN 5044
5043  GOTO 5050
5044  IF A$(8)=C$ THEN 5900
5050  IF A$(3)=A$(6) THEN 5052
5051  GOTO 5060
5052  IF A$(6)=A$(9) THEN 5054
5053  GOTO 5060
5054  IF A$(9)=C$ THEN 5900
5060  IF A$(4)=A$(5) THEN 5062
5061  GOTO 5070
5062  IF A$(5)=A$(6) THEN 5064
5063  GOTO 5070
5064  IF A$(6)=C$ THEN 5900
5070  IF A$(7)=A$(8) THEN 5072
5071  GOTO 5080
5072  IF A$(8)=A$(9) THEN 5074
5073  GOTO 5080
5074  IF A$(9)=C$ THEN 5900
5080  IF A$(3)=A$(5) THEN 5082
5081  GOTO 5990
5082  IF A$(5)=A$(7) THEN 5084
5083  GOTO 5990
5084  IF A$(7)=C$ THEN 5900
5085  GOTO 5990
5900  PRINT C$,"WINS THIS ROUND":W=1
5990  RETURN
6000  REM LINES 6100-6600 LOOK FOR A WINNING MOVE
6110  FOR I=1 TO 9
6120  IF A$(I)="X" THEN B(I)=1
6140  IF A$(I)="" THEN B(I)=0
6150  IF A$(I)="O" THEN B(I)=-1
6160  NEXT I
6200  READ I,J,K
6220  IF I=0 THEN 6500
6300 S=B(I) + B(J) + B(K)
6310  IF S=2 THEN 6400
6320  GOTO 6200
6400  IF B(J)=0 THEN A$(J)="X"
```

```
6405  GOTO 7000
6410  IF B(K)=0 THEN A$(K)="X"
6415  GOTO 7000
6420  IF B(I)=0 THEN A$(I)="X"
6425  GOTO 7000
6500  RESTORE
6510  READ I,J,K
6530  IF I=0 THEN 7100
6535 S=B(I) + B(J) + B(K)
6540  IF S=-2 THEN 6600
6550  GOTO 6510
6600  IF B(J)=0 THEN A$(J)="X"
6610  IF B(K)=0 THEN A$(K)="X"
6620  IF B(I)=0 THEN A$(I)="X"
7000  RESTORE : GOTO 4070
7100  RESTORE : GOTO 4000
8000  DATA 1,2,3,4,5,6,7,8,9,1,4,7,2
8010  DATA 5,8,3,6,9,1,5,9,3,5,7,0,0,0
```

Exercises (answer on page 322)

1. Modify the TIC TAC TOE program so that you and the computer may play a series of ten games. The computer should decide the champion of the series.
2. Modify the TIC TAC TOE program to play 4 x 4 tic tac toe.

ANSWER TO TEST YOUR UNDERSTANDING 1

See lines 120-170 of the program on page 247.

10.3 Application 10-2: Drawing Bar Charts

In this application program we will draw a bar chart to graphically represent some data. A bar chart merely uses the length of bars to indicate the frequency or count in various categories.

For example, if we wanted to graphically represent the number of persons with various years of service in an organization, this data would probably first be arranged in classes, also called categories or intervals. The data might look like this:

	years of service	count
category 1:	fewer than 2	12
category 2:	more than 2 through 5	21
category 3:	more than 5 through 10	37
category 4:	more than 10 through 15	8
category 5:	over 15	3

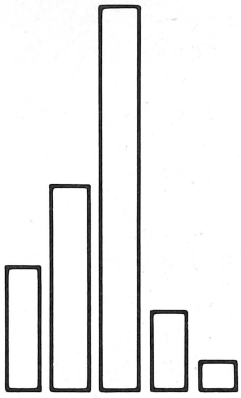


Figure 10-8. Bar chart for years of service.

We could depict this by a bar chart as shown in Figure 10-8.

We will write a program which will take in data either as a frequency distribution (already arranged in classes) or as raw data to be read in and summarized. The program will then display a bar chart for that data. Here is the solution procedure.

```
initialize: get the number of categories (maximum of 15)
            is data raw or arranged
            if raw, input upper bound for each category
                    input and count each piece of data
            if arranged, input count for each category
            find maximum frequency count
scale: scale frequency counts so that longest bar is 25
display: print out category number in bottom four lines
         enter low res graphics and plot each bar
```

The program is relatively straightforward except for the scaling. Here we wish to limit the height of the bars so that they fit on the screen. We have chosen to limit the bar height to 25. To scale each bar, we take the actual frequency count in that category divided by the maximum frequency count found. This tells us what proportion of the 25 cells should be plotted for that category. Next we multiply the proportion by 25 to get the actual number and then round and truncate using the INT function because we can only plot integral numbers of cells.

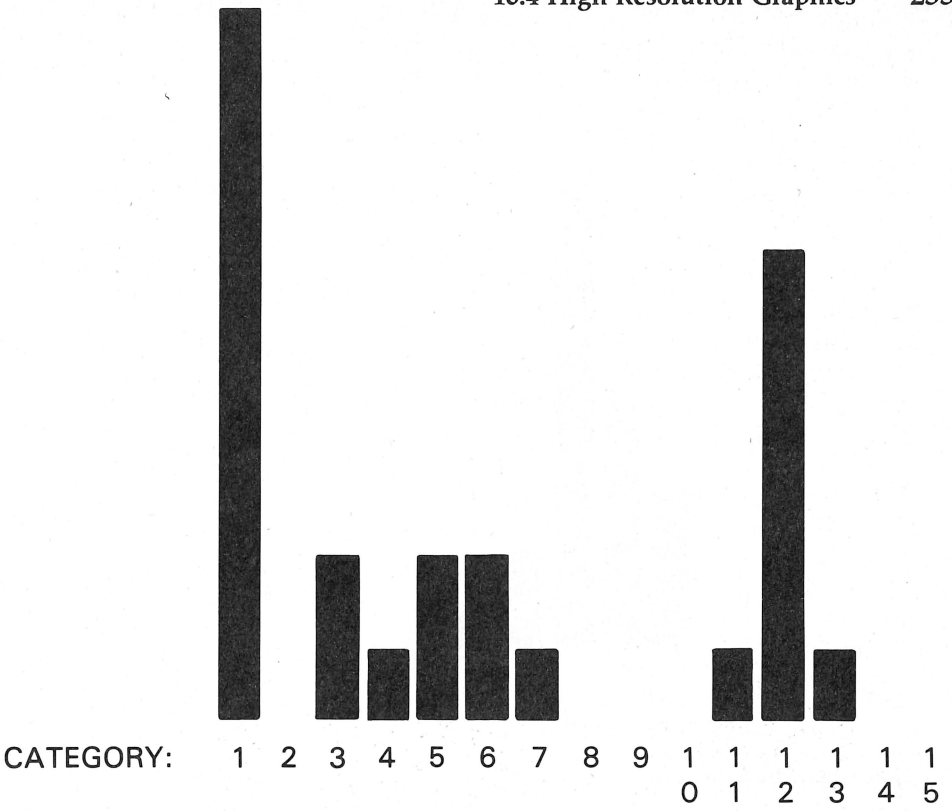
If we have the data already arranged into classes, the computer will expect to read the frequency counts in the DATA statements. If we are inputting raw data, the raw data will be placed in the DATA statements. We will show you an example run for each option. First, here is the program.

```
10 REM PROGRAM TO DRAW A BAR CHART
20 INPUT "NUMBER OF CATEGORIES (MAX 15)";C
30 IF C> 15 THEN PRINT "MAXIMUM OF 15 CATEGORIES":GOTO 20
40 INPUT "IS DATA ARRANGED OR RAW? (R/A)";A$:IF A$="A" THEN
        GOTO 200
```

```
50 DIM U(C),F(C),S(C): FOR I=1 TO C
60 PRINT "UPPER LIMIT FOR CATEGORY ";I;: INPUT U(I)
70 NEXT I
100 READ D: IF D=-1 THEN GOTO 300
110 FOR I=1 TO C-1
120 IF D <=U(I) THEN F(I)=F(I) + 1: GOTO 150
130 NEXT I
140 I=C
150 IF F(I) > MX THEN MX=F(I)
160 GOTO 100
200 FOR I=1 TO C
210 READ F(I)
220 IF F(I) > MX THEN MX=F(I)
230 NEXT I
300 FOR I=1 TO C
310 S(I)=INT(F(I)/ MX * 25 + .5)
320 NEXT I
330 HOME: GR: COLOR=15
340 VTAB(21): PRINT "CTRGY: ";: IF C < 10 THEN FOR I=1 TO
     C: PRINT I;" ";: NEXT I: PRINT
350 IF C >=10 THEN FOR I=1 TO 9: PRINT I;" ";: NEXT I:
          FOR I=10 TO C: PRINT "1 ";:NEXT I: PRINT
360 IF C >=10 THEN HTAB(27): FOR I=10 TO C: PRINT I-10;" ";:
          NEXT I: PRINT
370 FOR I=1 TO C
380 IF S(I)=0 THEN GOTO 400
390 VLIN 38,38 - S(I) AT 2*I +6
400 NEXT I
900 DATA 1,2,3,4,5,6,67,45,34,2,5,456,345,345,345,54,3,23,342,
          525,355,25,52,52235,45,235,-1
```

In this program, the data has been placed in line 900. It has been set up for entry of raw data. The output from this program is:

```
NUMBER OF CATEGORIES (MAX 15)15
IS DATA ARRANGED OR RAW? (R/A)R
UPPER LIMIT FOR CATEGORY 1?10
UPPER LIMIT FOR CATEGORY 2?20
UPPER LIMIT FOR CATEGORY 3?30
UPPER LIMIT FOR CATEGORY 4?40
UPPER LIMIT FOR CATEGORY 5?50
UPPER LIMIT FOR CATEGORY 6?60
UPPER LIMIT FOR CATEGORY 7?70
UPPER LIMIT FOR CATEGORY 8?80
UPPER LIMIT FOR CATEGORY 9?90
UPPER LIMIT FOR CATEGORY 10?100
UPPER LIMIT FOR CATEGORY 11?250
UPPER LIMIT FOR CATEGORY 12?500
UPPER LIMIT FOR CATEGORY 13?1000
UPPER LIMIT FOR CATEGORY 14?10000
UPPER LIMIT FOR CATEGORY 15?10000
```



Let's change the data in line 900 to represent data which has already been arranged in classes as in the years of service example. The data will look like

```
900 DATA 12,21,37,8,3
```

In this case we do not need a sentinel datum because we know precisely how many pieces of data we are reading. The output for this program is like that shown in the figure on page 256.

10.4 High Resolution Graphics

In principle, high resolution graphics on the Apple II Computer works similarly to low resolution graphics except that the screen is divided into 280 columns (0-279) and 192 rows (0-191). High resolution graphics mode is turned on with the instruction

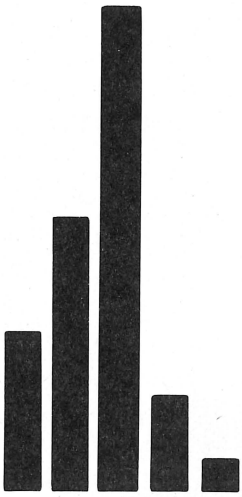
```
HGR
```

Again this gives us four lines of text at the bottom of the screen.

Color in high resolution mode is set via the instruction

```
HCOLOR=color number
```

```
NUMBER OF CATEGORIES (MAX 15)5
IS DATA ARRANGED OR RAW? (R/A)A
```



Program output.

Only eight colors are available in high resolution graphics. They are numbered 0-7. These colors are:

number	color
0	black1
1	green
2	blue
3	white1
4	black2
5	(depends on video display unit)
6	(depends on video display unit)
7	white2

All of these colors will depend somewhat on the characteristics of your particular video display unit.

In high resolution graphics, point plotting is accomplished via the instruction

```
HPLOT x,y
```

where x is the column number and y is the row number.

This command can also be used to draw lines. There are two ways to do this. The first assumes you have just plotted a point and instructs the computer to start at that point and draw a line to a second point. Its form is

```
HPLOT TO x,y
```

Another use for this command is in giving the two end points of a line. This form is

```
HPLOT x1,y1 TO x2,y2
```

The values of the points can be either numbers or numerical expressions within the acceptable range.

Let's use this command to draw a circle. If we let the center of the circle be placed at X0, Y0, and we let the radius be R, then our circle will look like Figure 10-9. A vertical line drawn straight through the middle of the circle will cross the circle at X0, Y0 - R at the top, and at X0, Y0 + R at the bottom. (Remember the Apple screen has rows numbered with 0 row at the top of the screen.) The circle is symmetric about this vertical line.

Let's say we wish to start drawing the circle at the bottom, draw the left side, then return to the bottom and draw the right side. Call any arbitrary point that we plot X1,Y1. As we plot successive points on either side, the value of Y1 will decrease. The value of X1 will change depending upon the value of Y1 chosen.

The equation of a circle centered at X0,Y0 is:

$$R^2 = (X1 - X0)^2 + (Y1 - Y0)^2$$

If we solve this equation for the value of X1, we obtain:

$$X1 = X0 \pm SQR(R^2-(Y1-Y0)^2)$$

As you can see, X1 will have two values: one for the left side (when we use the minus sign) and one for the right side of the circle.

To draw the circle let's begin by plotting a point at the bottom of the circle, or at X0,Y0+R. We then decrease Y1 by 1 and calculate the new value of X1 which will be

$$X1 = X0 - SQR(R^2-(Y1-Y0)^2)$$

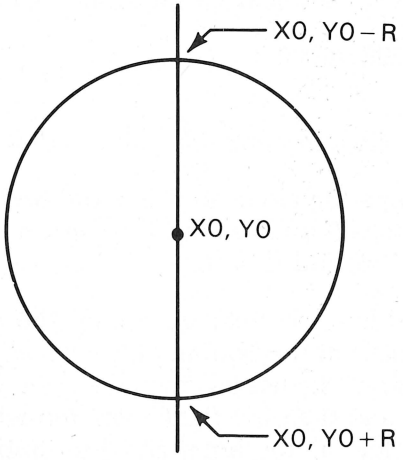


Figure 10-9.

For the left side the values of X1 will begin by getting smaller and then start to increase. We know we have finished drawing the left side when the value of X1 gets as large as the value of X0.

When we finish the left side, we start all over, drawing the right side. This time, though, the values of X1 will start by getting larger and then begin to diminish. We know we have finished the right side when the value of X1 gets as small as X0.

To change from subtracting the square root term to adding the square root term in calculating X1, we will use a variable called SN for sign. SN will begin as a -1 while we draw the left side. Then we will set it to a $+1$ for the right side. We will define a function to calculate the square root term.

Each time we draw a new point we will join it to the last point which was plotted. Here is the program.

```
 10 REM DEFINE CENTER AND RADIUS
 20 X0=60:Y0=60:R=25:RR=R^2:SN=-1
 30 DEF FN S(Y1)=SQR(RR - (Y1 - Y0)^2)
100 HGR:HCOLOR=3
110 X1=X0:Y1=Y0 + R:HPLOT X1,Y1
120 Y1=Y1 - 1:X1=X0 + SN*INT(FNS(Y1) + .5):
      HPLOT TO X1,Y1
130 IF X1>=X0 AND SN=-1 THEN SN=1:GO TO 10
140 IF X1<=X0 AND SN=1 THEN END
150 GO TO 120
```

When the program ends, you will still be in graphics mode. To get back into text simply type in the word

`TEXT` (or use the ESC - T sequence on the Apple IIe)

You could, of course, add the TEXT command to your program. Instead of ending at line 140, go to line 160. There you can build in a delay by using a FOR. . .NEXT loop that doesn't do anything. This allows time to look at your drawing. Then say TEXT. Here is what the additional lines might look like:

```
160 FOR I=1 TO 1000:NEXT I
170 END
```

Example 1. Add to the circle program so you can draw a little stickperson.

Solution. We will change the circle so it is more toward the middle of the screen and smaller. This will serve as the stickperson's head. We can do this by letting X0 = 90, Y0 = 30, and R = 10.

Instead of ending on line 140, let's go to line 200 where we can draw a stickperson body. We start at the bottom of the circle and draw a line down for the body. Next we draw the left leg, then the right. Finally, we go back up and draw the left arm and then the right arm. You will have to experiment with the length and angle of the arms and legs until you draw a picture which looks good to you. Here is a program which draws a stickperson picture to help you get started.

```
10  REM DEFINE CENTER AND RADIUS
20 XO=90:YO=30:R=10:RR=R^2:SN=-1
30  DEF FN S(Y1)=SQR (RR - (Y1 - YO) ^ 2)
100  HGR : HCOLOR=3
110 X1=XO:Y1=YO + R: HPLOT X1,Y1
120  Y1=Y1 - 1:X1=XO + SN * INT ( FN S(Y1) + .5):
        HPLOT TO X1,Y1
130  IF X1 >=XO AND SN=-1 THEN SN=1: GOTO 110
140  IF X1 <=XO AND SN=1 THEN GOTO 200
150  GOTO 120
200 HPLOT XO,YO + R TO XO, YO + 4 * R
210 HPLOT XO,YO + 4*R TO XO - R, YO + 6.5 * R
220 HPLOT XO,YO + 4*R TO XO + R, YO + 6.5 * R
230 HPLOT XO,YO + 2*R TO XO - R, YO + 4 * R
240 HPLOT XO,YO + 2*R TO XO + R, YO + 4 * R
```

Computer Art Revisited

Another method of generating art using the computer is to essentially "trace" a picture. Prepare a sheet of tracing paper the size of your picture. On this tracing paper lightly draw a grid with 280 columns and 160 rows. This tracing paper will be your video display worksheet.

For example, suppose that your picture is a photograph. Place a video display worksheet over the picture. Fill in all rectangles which touch the subject of the photograph. In this way, you will create an impression of the subject. By plotting the appropriate points on the high resolution screen, you will create a computer photo. This is a tedious process. In Chapter Thirteen we will discuss some peripheral equipment which can be used to simplify this process.

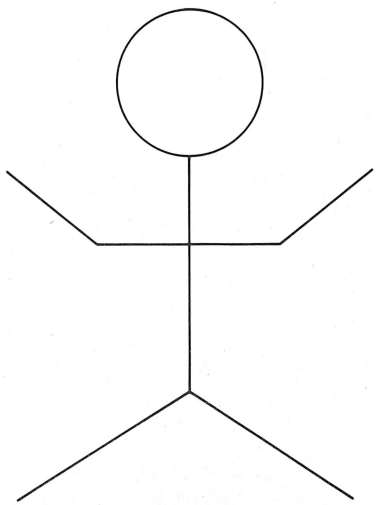


Figure 10-10.

Exercises

1. Modify the program given in Example 1 to draw a jumping stickperson. The picture should look something like Figure 10-10.
2. Create a computer impression of a member of your family using a large photograph. (5″ x 7″ or larger will work best. Use high resolution graphics.)

10.5 Application 10-3: Drawing Line Graphs

Frequently, we are faced with the problem of analyzing data which has been collected in regular time periods. For example, we might want to take the data that we collected in the expense tracking program of Section 7.3 to see whether there is any trend (an increase or decrease over time) or any seasonality (differences in expense level which we can attribute to seasonal factors, like a heating bill in winter months).

One of the easiest ways to get a feel for whether such factors are important in your data is to plot it. In this section we will write a program to take data which has been observed on some regular time basis and draw a line graph for that data. A line graph plots the points, joining every two consecutive points with a line segment.

Assume we have monthly gas bills for a small apartment building. Our data is as follows:

January	2483.42
February	1064.32
March	1241.20
April	747.52
May	382.62
June	208.94
July	203.07
August	222.13
September	262.27
October	723.48
November	1041.11
December	2580.83
January	2788.69
February	1212.48
March	2365.37
April	1149.48
May	1574.91
June	332.76
July	263.41
August	279.18
September	356.99
October	1031.54
November	1538.47
December	1834.65

In order to draw a line plot of this, we will use the HPLOT function in high resolution graphics. Here is the solution structure:

```
initialize: is it monthly, quarterly or yearly data?
            get number of periods to be plotted ( max 24)
            get data
            find maximum datum
plot: scale data (we will use 150 rows for our graph)
      draw axes
      plot line segments
      print comments
```

The program can be written directly from the solution structure. We will start the axes at column 20 and row 150. Remember that the row and column numbers start with 0 in the upper left.

To scale the data so that it fits into the 150 rows, we find the maximum datum, divide it into each individual value to get the proportion of the space to be allotted to the value, then multiply by 150 to translate that into absolute number of rows.

Here is the program.

```
10 HOME: VTAB(5); PRINT "IS DATA TO BE PLOTTED "
20 PRINT " " : PRINT "MONTHLY (M)"
30 PRINT " " : PRINT "QUARTERLY (Q)"
40 PRINT " " : PRINT "YEARLY (Y)"
50 PRINT " " : PRINT "TYPE IN ONE LETTER ";:GET A$: PRINT A$
60 IF A$="M" THEN FC=12: GOTO 100
70 IF A$="Q" THEN FC=4: GOTO 100
80 IF A$="Y" THEN FC=1: GOTO 100
90 PRINT " " : PRINT SPC(10);"M, Q, OR Y?": VTAB (12):GOTO 50
100 VTAB(15): INPUT "HOW MANY PERIODS OF DATA (MAX 24)? ";P
110 IF P> 24 THEN VTAB(17): PRINT SPC(10);"MAXIMUM OF 24":
          VTAB(15); GOTO 100
150 DIM D(P),S(P)
160 FOR I=1 TO P
170 READ D(I)
180 IF D(I) > MX THEN MX=D(I)
190 NEXT I
200 FOR I=1 TO P
210 S(I)=(D(I) / MX) * 150
220 NEXT I
500 DATA 2483.42,1064.32,1241.20,747.52,382.62,208.94
510 DATA 203.07,222.13,262.27,723.48,1041.11,2580.83
520 DATA 2788.69,1212.48,2365.37,1149.48,1574.91,332.76
530 DATA 263.41,279.18,356.99,1031.54,1538.47,1834.65
1000 HGR: HCOLOR=3
1010 HPLOT 20,0 TO 20,170
1020 HPLOT 0,150 TO 270,150
1030 FOR I=30 TO 260 STEP 10: HPLOT I,149 TO I,151: NEXT I
1040 IF FC <> 1 THEN FOR I=20 TO 260 STEP 10 * FC:
          HPLOT I,147 TO I,153: NEXT I
```

```
2000 HPLOT 30, S(1)
2010 FOR I=2 TO P
2020 HPLOT TO 20 + 10 * I, S(I)
2030 NEXT I
```

Run the program with the data provided. Then see how it works using some data of your own.

Exercises

1. Incorporate the line graph into your monthly expense program of Section 7.3 (page 174). Since that program only takes in one year of monthly data, you will probably want to suppress the menu and change the scale on the horizontal axis so that each point is plotted in 20 columns instead of ten.

10.6 Forming and Saving Shapes

There is quite a bit you can do with high resolution graphics, but some of the most interesting applications involve using the ability to draw figures and move them about the screen. Unfortunately, using this feature of the Apple involves getting a litle more deeply into the mechanics of the Apple II than interests most people.

In this section we will introduce the principles of forming and saving such figures, called shapes. There are several software packages on the market which facilitate this process; these will be discussed in Chapter Twelve.

In Section 8.1 we discussed the idea of bits and bytes. We said that a bit was just a 0/1 signal used to code all pieces of data and all instructions which the computer used. A byte was a set of eight bits. The Apple computer is organized in bytes so that if you looked at a particular memory location you would see some representation of these eight bits. We did not dwell on these facts because they are not necessary to know if you are using Applesoft BASIC. However, we do have to use them a little if we are going to draw and save shapes. We will take you through the principles of doing this step by step to make it as simple as possible.

The basic idea is that we wish to develop what is called a shape table. This is a table of codes for different shapes we may wish to use in high resolution graphics programs. Such tables may be saved on disk in binary code.

In Chapter Seven we introduced random and sequential data files. We can also create files which store data in binary code. These are called **binary files** and differ from the program and data files we have discussed thus far. We will store the codes for different shapes in such a binary file. This file of shape codes is called a **shape table**.

To illustrate the idea of a shape table, we will start by developing a table with one shape. Later, we will add another shape to it.

The first step in developing a shape table is drawing a shape. Using some graph paper as a video display work sheet, let's draw a fir tree. Start at the upper right corner of one of the squares on the grid and draw your picture using only the edges of the squares, as in Figure 10-11.

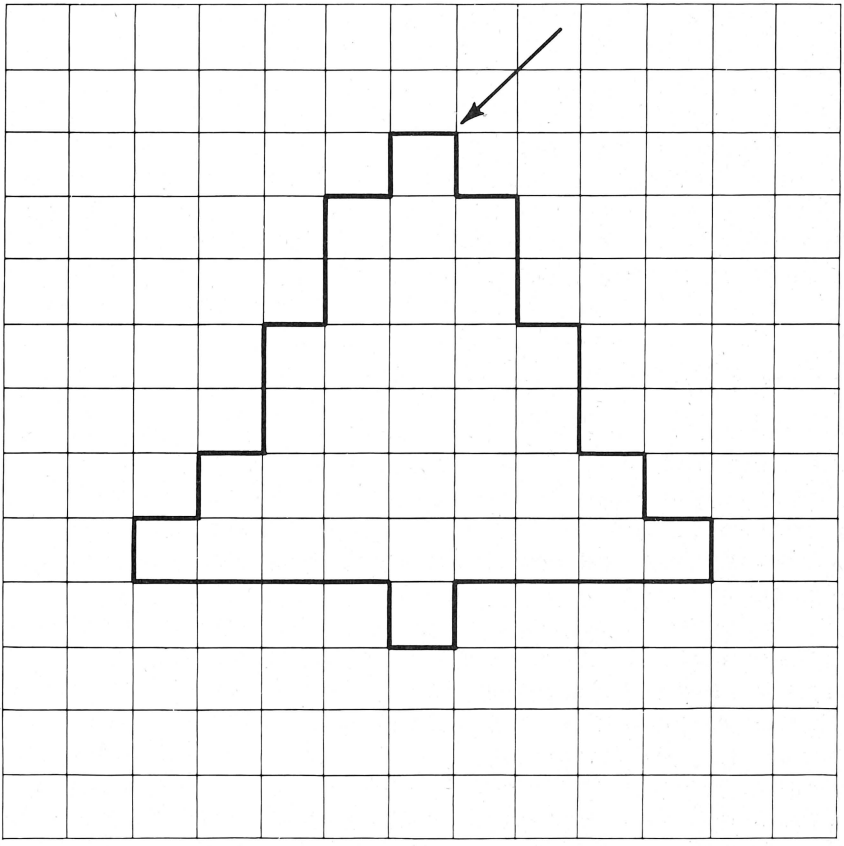


Figure 10-11. A fir tree.

In this picture we could start at one point and draw the entire tree without ever taking our pencil from the paper. This is not a necessary feature of high resolution shapes as we shall see with the next shape we draw, but it does make things a little easier for our first attempt.

Starting at some point on the picture, you must record information about each line drawn. This information includes whether you are plotting the line (in this case that's all we are doing), whether you are just moving to another corner on the grid, or whether you are moving down, up, right, or left.

Let's start at the upper right corner of our picture. Write down a one (1) if you are plotting or a zero (0) if you are just moving. Then record the direction (D, U, L, or R). Our sequence of moves to draw the fir tree is (note: in this example, we always have a 1 because we are always plotting):

```
code    binary    byte

 1D       110     00101110
 1R       101
 1D       110     00110110
 1D       110

 1R       101     00110101
 1D       110
 1D       110     00101110
 1R       101

 1D       110     00101110
 1R       101
 1D       110     00111110
 1L       111

 1L       111     00111111
 1L       111
 1L       111     00110111
 1D       110

 1L       111     00100111
 1U       100
 1L       111     00111111
 1L       111

 1L       111     00111111
 1L       111
 1U       100     00101100
 1R       101

 1U       100     00101100
 1R       101
 1U       100     00100100
 1U       100

 1R       101     00100101
 1U       100
 1U       100     00101100
 1R       101

 1U       100     00101100
 1R       101
                  00000000
```

Figure 10-12. Codes for drawing a fir tree shape.

Now each line drawn has to be converted to a three-digit binary code. We have already provided for this by using a 1 for plotting and a 0 for moving. Now we will substitute the following binary codes for the U, D, L, and R:

```
U=00
D=10
L=11
R=01
```

This is shown next to the letter codes in the fir tree table.

At this point we have each line in our picture coded as three-digit binary numbers. When we store these codes in a binary file, we will be storing them as bytes (sets of eight bits). Now we will combine these three-digit sets into eight-bit bytes.

To do this we take the first set of three bits and put them in the rightmost bits of the byte:

```
          1 1 0
- - - - - - - - -
```

The next three bits go immediately next to this:

```
      1 0 1 1 1 0
- - - - - - - - -
```

There are two spaces left in the byte at the left. The next set of three bits starts with a 1, so it will not fit into the two empty spaces. Therefore, we fill these empty spaces with zeroes. The finished byte looks like this:

```
0 0 1 0 1 1 1 0
- - - - - - - - -
```

We continue through the fir tree in this fashion. Whenever we are able to fit a three-bit code into the leftmost two spaces (whenever it happens to be a non-plotting move), we do so simply by ignoring the leading zero (After all, the zero in front of a number does not affect the value of the number). Otherwise, we fill the spaces with zeroes and move to the next byte. We always signify the end by adding one byte which contains only zeroes. The byte codes are shown in Figure 10-12.

We must now insert these codes into memory in the format required for an Apple II shape table. The shape table required consists of two parts: a **header area** which describes the number of shapes in the table and the starting location of the codes for each shape, and a **shape definition** area which contains the byte codes.

Location 0 of the table always contains the number of shapes. Location 1 is always equal to 0 (actually it is ignored and can be anything). Locations 2 and 3 contain the location of the first shape definition, 4 and 5 contain the second definition, and so on. The length of this header section depends on the number of shapes. If SH represents the number of shapes, then the header section will require 2 + 2*SH locations. For our one-shape table, the header section will contain:

```
location 0--number of shapes (1)
         1--0
         2 } location of first definition
         3 }   in table (4)
```

The location of the definitions are stored in two locations with the low order bits in the first location, and the high order bits in the second.

Our complete shape table translated into binary, hexidecimal, and decimal codes, respectively, is:

```
                binary    hex  decimal
location  0:    00000001   01      1
          1:    00000000   00      0
          2:    00000100   04      4
          3:    00000000   00      0
          4:    00101110   2E     46
          5:    00110110   36     54
          6:    00110101   35     53
          7:    00101110   2E     46
          8:    00101110   2E     46
          9:    00111110   3E     62
         10:    00111111   3F     63
         11:    00110111   37     55
         12:    00100111   27     39
         13:    00111111   3F     63
         14:    00111111   3F     63
         15:    00101100   2C     44
         16:    00101100   2C     44
         17:    00100100   24     36
         18:    00100101   25     37
         19:    00101100   2C     44
         20:    00101100   2C     44
         21:    00000000   00      0
```

As you can see, the length of this shape table is 22 bytes numbered from 0 through 21.

We can now put this table in memory and proceed to use it to draw the fir tree. There are a few ways in which we can place this in memory. We will use the only method which we have learned, and that is by POKEing into the appropriate memory locations.

What are the appropriate memory locations? The Apple II reserves certain portions of memory for your BASIC programs. We want to be sure that we don't POKE our shape table into these locations. If we were to do that we would destroy parts of the program. A discussion of how memory is allocated to different functions is beyond the scope of this book, but if we use memory locations starting at location 3328 (D00 in hexadecimal), we will be safe. Here is a short program which will accomplish the POKE process for our fir tree.

```
100 FOR I=0 TO 21
110 READ D
120 POKE 3328 + I, D
130 NEXT I
900 DATA 1,0,4,0,46,54,53,46,46,62,63,55,39,63,63,
     44,44,36,37,44,44,0
```

We are POKEing the decimal representation of our shape table into successive memory locations starting at location 3328.

Type in the above program and RUN it. When you have done this, we will almost be ready to use this shape. At this point we have defined the shape table and entered it into memory. We have a few more steps left. We must

tell the computer where the shape table is, enter high resolution graphics, set the parameters for the hi-res graphics, and DRAW the shape.

Your Apple II expects to find the starting location of the shape table at locations 232 and 233 with the high order digits in location 233. Since our table starts at location 3328 (D00 in hexadecimal), we must POKE the value of 0 in location 232 and the value of 13 (D in hex) in 233. You can do this either in immediate or in deferred mode. For now let's do this in immediate mode by typing in

```
POKE 232,0
POKE 233,13
```

Now we can go into high resolution graphics mode and set the parameters. To enter hi-res graphics, we need only type in

```
HGR
```

The parameters which must be set include the color, the scale, and the rotation. For the time being, let's use color number 3 by typing in

```
HCOLOR=3
```

The scale can be set from 1 to 255. A scaling factor of 1 is normal size, or point by point reproduction of the shape. Let's type in

```
SCALE=1
```

The rotational angle at which the shape is pictured can be set at any value from 0 to 255. A value of 0 will cause it to be drawn exactly as the shape was defined. A value of 16 causes a 90 degree rotation, 32 is 180 degrees and so on. Let's start with

```
ROT=0
```

We have entered high resolution graphics and set the parameters. Now we are ready to look at our picture. To draw the shape we can use the DRAW command. Its form is

```
DRAW shape AT x,y
```

where **shape** is the number of the shape in the table (in our case this is 1 because our table only has one shape), x is the high resolution row (0 to 279 where 0 is the top row), and y is the high resolution column (0 to 159) where 0 is at the left. Let's draw our shape at 60,60. To do this, type in

```
DRAW 1 AT 60,60
```

You should finally see the picture of the fir tree appear on the screen.

TEST YOUR UNDERSTANDING 1 (answers on page 272)

a. Draw the fir tree at location 120,90.
b. Change the scale to 2 and draw the tree at location 120,30.
c. Rotate the figure 90 degrees and draw at 40,140.

Now that we have the shape table in memory, let's save it on disk so we don't have to re-enter it each time. To do this we want to save a binary file. This is done via the BSAVE command. The form of this command is

```
BSAVE name, Aaddress, Llength
```

where **name** is the name we wish to give the table when it is saved on disk, **address** is the location of the table in memory, and **length** is the number of bytes in the table. For our fir tree the command would be

```
BSAVE FIR, A3328, L22
```

We can load this same shape table back into memory using the BLOAD command. The form of this command is

```
BLOAD name, Aaddress
```

where **name** is the name of the table and **address** is the memory location into which we wish to put the table. If you wish to load the table into the same location from which you saved it, you do not have to specify the address parameter. We just say

```
BLOAD FIR, A3328
```

or simply

```
BLOAD FIR
```

As you can see, developing a shape table can be quite tedious. To make this a little bit easier for you, we have developed a program which will take your shapes coded as 0 or 1 plus U, D, L, or R and convert them to decimal coded bytes, set up the shape table, POKE it into memory, then BSAVE the table under a name of your choice.

The program assumes that you have put your coded shapes into DATA statements. This will allow you to edit the DATA statement and the shape if you are not satisfied with it. The listing below includes the definition for the fir tree. Each shape definition should end with a 9.

Here is the program which creates the shape table. We will use it in Example 1 to create and save a smiling face as a shape table. All you need do to use the program is define your shape first.

```
5 POKE 232,0: POKE 233,13
10 HOME: INPUT "NUMBER OF SHAPES? ";SH
20 POKE 3328,SH; POKE 3329,0
30 D1=2 * SH + 2: D=3328 +D1
40 FOR S=1 TO SH
41 IF SH=0 THEN TEXT: HOME: PRINT "NO SHAPES TO SAVE": END
42 IF S > SH THEN POKE 3328,SH: GOTO 5000
45 B1=B + D1: BH=B: H=INT (B1/256): LO=B1- 256*H
50 POKE 3328 + 2*S,LO: POKE 3329 + 2*S,H
60 VTAB(21); I=0: FLASH: PRINT "DEFINING SHAPE":NORMAL
70 READ Z$; A=VAL(Z$)
80 IF A=9 THEN GOTO 3000
90 IF I=2 AND A=1 THEN GOSUB 2000
100 A$=RIGHT$(Z$,1): GOSUB 1000
110 L=(4*A+A1) * 8^I + L
120 I=I+1: IF I=3 THEN GOSUB 2000
130 GOTO 70
```

```
1000 IF A$="U" THEN A1=0
1010 IF A$="D" THEN A1=2
1020 IF A$="L" THEN A1=3
1030 IF A$="R" THEN A1=1
1040 RETURN
2000 POKE D + B, L
2010 L=0: B=B+1: I=0
2020 RETURN
3000 IF L <> 0 THEN GOSUB 2000
3010 GOSUB 2000
3020 GOSUB 4000
3030 NEXT S
3040 GOTO 5000
4000 HGR: ROT=0: HCOLOR=3
4010 FOR I=1 TO 3
4020 SCALE=1
4030 DRAW S AT 60 + (I-1) * 60,60
4040 NEXT I
4050 VTAB(21): PRINT "SAVE THIS SHAPE? (Y/N) ";: GET Z$:
          PRINT Z$
4060 IF Z$="N" THEN B=BH: S=S-1: SH=SH-1
4070 RETURN
5000 INPUT "NAME OF THIS SHAPE TABLE? ";:N$
5010 PRINT CHR$(4);"BSAVE";N$;",A3328,L";B
5020 TEXT: HOME: END
9000 DATA 1D,1R,1D,1D,1R,1D,1D,1R,1D,1R,1D,1L,1L,1L,1L,1D,
          1L,1U,1L,1L,1L,1L,1U,1R,1U,1R,1U,1U,1R,1U,1U,1R,
          1U,1R,9
```

Example 1. Define a shape of a smiling face and use the above program to save it as a shape table.

Solution. Here are the steps we will need:

1. Draw the shape on graph paper (your video display worksheet).
2. Code the shape by translating each line into a 1 if it is a plotting line, or a 0 if it is simply a move without a plotting line, and a U, D, L, or R depending on whether you are moving up, down, left, or right.

   CAUTION: In deciding how you will move about the sheet, there is one rule which you must follow. You may never make two consecutive moves in the up direction without plotting. When these are coded into the table, they will appear as zeroes and the computer will think you have finished the shape!!!
3. Enter the codes into DATA statements and RUN the program given above to enter the shape table and save it under the name SMILE.

Here is the smiling face drawn on graph paper. (See Figure 10-13.) You are free to draw a more handsome face if you wish!

Now let's code this face. Remember we cannot make two consecutive moves up without plotting. There are many ways to do this. We will start at the arrow in Figure 10-14 and follow the dashed line as we move around the face.

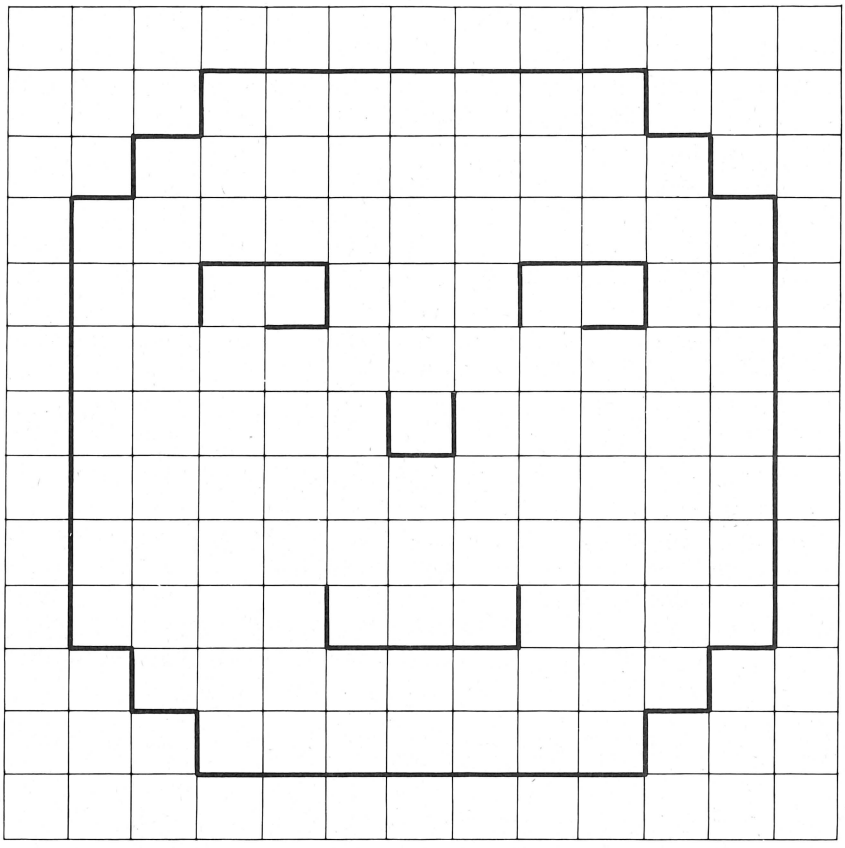
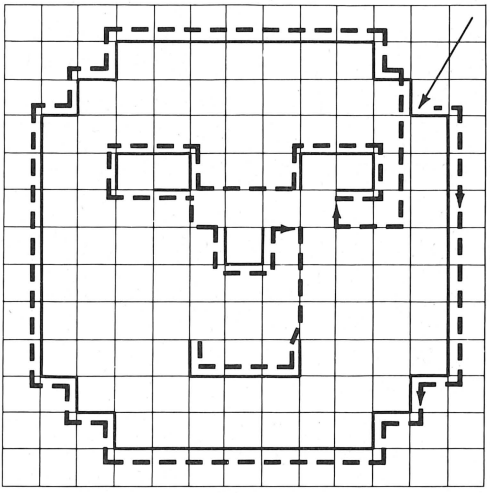


Figure 10-13. A smiling face.

If you examined the computer's rendition of the face as we coded it, you probably noticed that the nose and mouth were a little off. This is a problem with drawing high resolution shapes because we are not exactly drawing lines but plotting points. Sometimes the picture comes out slightly different from what we had planned. The easiest thing to do is to examine the picture and correct the coding. You can do this easily in your data statements.

There is another command for drawing high resolution shapes which is sometimes very valuable. This is the XDRAW command. When we use the DRAW command, our shape is drawn on the high resolution screen at the point specified and in the color specified. With the **XDRAW** command, our first request to XDRAW will put the shape where we say, but the second request to XDRAW will draw the shape in the background color. This is equivalent to erasing the shape. In this way we can cause the shape to blink.

If you have just run the smiling face program, the shape table will be in memory and all the parameters will be set. If not, do the following:



```
1R  1R  1L
1D  1U  1D
1D  1R  0R
1D  1U  1R
1D  1R  0D
1D  1R  0R
1D  1R  1D
1D  1R  1R
1L  1R  1U
1D  1R  0R
1L  1R  0D
1D  1D  0D
1L  1R  0D
1L  1D  1D
1L  0D  1L
1L  0D  1L
1L  0D  1L
1L  0L  1U
1L  0L
1U  0U
1L  1R
1U  1U
1L  1L
1U  1L
1U  1D
1U  0L
1U  0L
1U  0L
1U  1U
1U  1L
```

Figure 10-14. Coding the smiling face.

```
BLOAD SMILE
POKE 232,0
POKE 233,13
HGR
HCOLOR=3
ROT=0
SCALE=1
```

Now, XDRAW the face several times in the same location using the following program:

```
10 FOR I=1 TO 25
20 XDRAW 1 AT 60,60
30 NEXT I
```

The face will seem to flash on and off. It is actually being drawn repeatedly, first in color 3, then in the background color. We will use this in Section 10.8 to illustrate some animation techniques.

TEST YOUR UNDERSTANDING 2 (answer on page 272)

Modify the flashing face program so that the face flashes slower.
Hint: Use a FOR. . .NEXT loop with nothing in it.

As your efforts at creating shapes and working with high resolution graphics become more ambitious, you may begin to run into problems because of the way that the Apple II allocates memory space. Discussion of this is beyond the scope of this book. If you are going to use a lot of high resolution graphics features, you insert the following line in your programs:

```
LOMEM: 24600
```

This separates your program area from the area used by the high resolution screens. For further discussion of this topic, see your Applesoft manual.

Exercises

1. Develop a shape table containing two shapes. Let them both be stickmen rowing a boat, but in the first shape let the man have the oars forward, and in the second, let him have the oars in back.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1 and 2

1: a.
```
DRAW 1 AT 120,90
```
b.
```
SCALE=2
DRAW A AT 120,30
```
c.
```
ROT=16
DRAW 1 AT 40,140
```
2: Add the following line:
```
25 FOR I=J TO 500: NEXT J
```

10.7 Animation Principles

The area of graphics and animation is a fascinating one. We are only touching the surface in this book because it is also a rather complicated area. If you have been following along up to now, you are probably ready to learn a little about animation. Basically, there are two techniques which can be used to obtain some animation on your Apple II computer.

The Apple II computer has two high resolution areas in memory called high resolution screen, or page 1, and high resolution screen, or page 2. When you say HGR to get into high resolution graphics, the screen you see and upon which you draw your pictures is high resolution screen 1. There is a second screen identical to the first in size and capability. To access this

second high resolution screen, we can say HGR2. One of the techniques used to animate figures is to draw pictures on each of the screens and then quickly switch between the two. This is similar to drawing pictures on each page of a small pad of paper such that each picture represents what the scene might look like a few seconds later. When you flip through the pages quickly, it appears that the scene is actually moving.

The second technique is to use only one screen but to draw and redraw a picture in different places on the screen. This gives the appearance that the figure is moving about the screen.

In this section and the next we will build on the material we developed in previous sections in order to illustrate these two techniques. We will also learn a few more important memory locations into which we can POKE useful information.

Let's begin with the two-screen approach. We will use the stickperson developed in Section 10.4 and the one which was assigned in one of the exercises for that section. The figure which we drew as part of the material was a standing figure; the stickperson drawn for the exercise was a jumping figure. We will put them both together so that the stickperson actually appears to be jumping.

First, we need to draw the stick figures on the two high resolution screens. We will draw the jumping figure on screen 1 and the standing figure on screen 2. To facilitate the process we will use subprograms, drawing the head in subprogram 100, the jumping body in subprogram 200, and the standing body in subprogram 300. We have not made any major changes to the programs developed earlier. Here is the combined program.

```
10 REM DEFINE CENTER AND RADIUS
20 XO=90:YO=30:R=10:RR=R^2:SN=-1
30 DEF FN S(Y1)=SQR (RR - (Y1 - YO)^2)
40 HGR: GOSUB 100: GOSUB 200
50 SN=-1: HGR2: GOSUB 100: GOSUB 300
70 END
100  REM DRAW A HEAD
110 X1=XO:Y1=YO + R: HPLOT X1,Y1
120  Y1=Y1 - 1:X1=XO + SN * INT ( FN S(Y1) + .5):
          HPLOT TO X1,Y1
130  IF X1>=XO AND SN=-1 THEN SN=1: GOTO 110
140  IF X1<=XO AND SN=1 THEN RETURN
150  GOTO 120
200 HPLOT XO,YO + R TO XO, YO + 4 * R
210 HPLOT XO,YO + 4*R TO XO - 2* R, YO + 6 * R
220 HPLOT XO,YO + 4*R TO XO + 2* R, YO + 6 * R
230 HPLOT XO,YO + 2*R TO XO - R, YO + 2 * R
240 HPLOT TO XO - 2*R, YO + .5*R
250 HPLOT XO,YO + 2*R TO XO + R, YO + 2 * R
260 HPLOT TO XO + 2*R, YO + .5*R
270 RETURN
300 HPLOT XO,YO + R TO XO, YO + 4 * R
310 HPLOT XO,YO + 4*R TO XO - R, YO + 6.5 * R
320 HPLOT XO,YO + 4*R TO XO + R, YO + 6.5 * R
```

```
330 HPLOT XO,YO + 2*R TO XO - R, YO + 4 * R
340 HPLOT XO,YO + 2*R TO XO + R, YO + 4 * R
350 RETURN
```

This program merely draws the figures on the two screens. The figures are drawn so the heads and bodies are in the same place on both screens. Now we want to look at one screen after the other in succession.

First, we need to expand our knowledge of memory locations to include some which are useful when dealing with high resolution graphics. You notice that when you say HGR, you automatically obtain a clear high resolution screen with four lines of text visible at the bottom. When you say HGR2, you switch to a clear screen with no text lines. We want to do two things. First, we would like to make sure that we have a full graphics screen. Then we want to switch from one screen to the other **without clearing** the screen.

Assuming that we are already in graphics mode, we can switch from graphics + four lines of text to full graphics by using

```
POKE -16302,0
```

You can poke this value when you are in text mode, but you will not see any results until you enter graphics mode. You can get back into mixed text + graphics mode by using

```
POKE -16301,0
```

Assuming you are in graphics mode, you can switch from one screen to the other without clearing the screens by using the following:

```
POKE -16300,0        to get to page 1
POKE -16299,0        to get to page 2
```

These commands will also work in text mode and in low resolution mode. Page 2 in these modes is difficult to access, so we will not really want these commands in anything except high resolution graphics.

Now let's modify our program so that we can alternately look at the two high resolution screens. Because we drew the jumping man on screen 1 and then accessed the second screen by using HGR2, we will already be in full screen graphics. All we need to do then is to make the screen switch.

To do this we will want to alternately poke 0 into locations -16300 and -16299. Let's let P1=-16300 because that's the location we need for page 1. Similarly, let P2=-16299. We will want to add a little delay between screens; otherwise, they will move so quickly we will not be able to see what is happening.

We will initialize P1 and P2 at line 30 and then add line 60 to make the switch 50 times with a delay of 200 (S) built in between each switch. Here is the new program.

```
10  REM DEFINE CENTER AND RADIUS
20 XO=90:YO=30:R=10:RR=R^2:SN=-1
30  DEF FN S(Y1)=SQR (RR - (Y1 - YO)^2)
40 HCOLOR=3: P1=-16300: P2=-16299
50 HGR: GOSUB 100: GOSUB 200
```

```
60 SN=-1: HGR2: GOSUB 100: GOSUB 300
70 S=200: FOR I=1 TO 100: POKE P1,0: FOR J=1 TO S: NEXT J:
          POKE P2,0: FOR J=1 TO S: NEXT J: NEXT I
80 END
100  REM DRAW A HEAD
110 X1=XO:Y1=YO + R: HPLOT X1,Y1
120  Y1=Y1 - 1:X1=XO + SN * INT ( FN S(Y1) + .5):
          HPLOT TO X1,Y1
130  IF X1 >=XO AND SN=-1 THEN SN=1: GOTO 110
140  IF X1 <=XO AND SN=1 THEN RETURN
150  GOTO 120
200 HPLOT XO,YO + R TO XO, YO + 4 * R
210 HPLOT XO,YO + 4*R TO XO - 2* R, YO + 6 * R
220 HPLOT XO,YO + 4*R TO XO + 2* R, YO + 6 * R
230 HPLOT XO,YO + 2*R TO XO - R, YO + 2 * R
240 HPLOT TO XO - 2*R, YO + .5*R
250 HPLOT XO,YO + 2*R TO XO + R, YO + 2 * R
260 HPLOT TO XO + 2*R, YO + .5*R
270 RETURN
300 HPLOT XO,YO + R TO XO, YO + 4 * R
310 HPLOT XO,YO + 4*R TO XO - R, YO + 6.5 * R
320 HPLOT XO,YO + 4*R TO XO + R, YO + 6.5 * R
330 HPLOT XO,YO + 2*R TO XO - R, YO + 4 * R
340 HPLOT XO,YO + 2*R TO XO + R, YO + 4 * R
350 RETURN
```

Now let's assume we do indeed want to put a comment on the picture, so we want partial text. We can do that by clearing the text screen (using HOME), printing our label at the desired row (we will only have the last four), and then going into mixed mode (POKE -16301,0) at the appropriate time. Line 15 below will print out our message.

```
15 HOME: VTAB(21): HTAB (5): PRINT "THIS MAN CAN
     REALLY JUMP!!"
```

When we print this line, it will be printed on text page 1 and not on page 2. We will want to be in mixed mode when we look at page 1 of high resolution graphics and in full screen graphics when we look at page 2. We can switch between mixed and full graphics using the following:

```
POKE -16301,0        to get to mixed mode
POKE -16302,0        to get to full graphics
```

When we look at page 1, we will want to poke a 0 in -16301; thus, we will let MX=-16301. When we move to screen 2, we will want full graphics; thus, we will let FL=-16302.

Here is the complete program with the new line of text at the bottom.

```
10  REM DEFINE CENTER AND RADIUS
20 XO=90:YO=30:R=10:RR=R^2:SN=-1
30  DEF FN S(Y1)=SQR (RR - (Y1 - YO)^2)
35 HOME: VTAB(21): HTAB(5): PRINT "THIS MAN CAN REALLY
   JUMP!!"
```

```
40 HCOLOR=3: P1=-16300: P2=-16299: MX=-16301: FL=-16302
50 HGR: GOSUB 100: GOSUB 200
60 SN=-1: HGR2: GOSUB 100: GOSUB 300
70 S=200: FOR I=1 TO 100: POKE MX,0: POKE P1,0: FOR J=1 TO
     S: NEXT J: POKE FL,0: POKE P2,0: FOR J=1 TO S: NEXT
     J: NEXT I
80 END
100  REM DRAW A HEAD
110 X1=X0:Y1=Y0 + R: HPLOT X1,Y1
120  Y1=Y1 - 1:X1=X0 + SN * INT ( FN S(Y1) + .5):
          HPLOT TO X1,Y1
130  IF X1 >=X0 AND SN=-1 THEN SN=1: GOTO 110
140  IF X1 <=X0 AND SN=1 THEN RETURN
150  GOTO 120
200 HPLOT X0,Y0 + R TO X0, Y0 + 4 * R
210 HPLOT X0,Y0 + 4*R TO X0 - 2* R, Y0 + 6 * R
220 HPLOT X0,Y0 + 4*R TO X0 + 2* R, Y0 + 6 * R
230 HPLOT X0,Y0 + 2*R TO X0 - R, Y0 + 2 * R
240 HPLOT TO X0 - 2*R, Y0 + .5*R
250 HPLOT X0,Y0 + 2*R TO X0 + R, Y0 + 2 * R
260 HPLOT TO X0 + 2*R, Y0 + .5*R
270 RETURN
300 HPLOT X0,Y0 + R TO X0, Y0 + 4 * R
310 HPLOT X0,Y0 + 4*R TO X0 - R, Y0 + 6.5 * R
320 HPLOT X0,Y0 + 4*R TO X0 + R, Y0 + 6.5 * R
330 HPLOT X0,Y0 + 2*R TO X0 - R, Y0 + 4 * R
340 HPLOT X0,Y0 + 2*R TO X0 + R, Y0 + 4 * R
350 RETURN
```

10.8 Application 10-4: Rowing Across The Screen

In this section we will develop a program using the second animation technique: repeatedly XDRAWing a picture on the screen.

In Section 10.6 one of the exercises asked you to define a shape table consisting of a man rowing a boat. The first shape had the man with the oars thrust forward, and the second had the man with the oars in back. Here are what the data statements in the shaper program might have looked like (if yours are somewhat different, that merely means that we drew the shape differently or moved around it in a different sequence):

```
9000 DATA 1L,1L,1L,1L,1L,1L,1L,1L,1D,0R,1R,1D,1R,1D,1R,1R,
          1R,1R,1R,1R,1R,1R,1R,1R,1U,1R,1U,1R,1U,1R,0U,1L,
          1L,1L,1L,1L,1L,1L,1U,1U,1U,1U,1U,1R,1U,1U,1L,1L,
          1D,1D,0D,0D,1L
9010 DATA 1L,1L,0U,0L,1D,0R,1D,0R,1D,0R,1D,0R,1D,0R,1D,0R,
          1D,0R,1D,0R,1D,0R,1R,9
9020 DATA 1L,1L,1L,1L,1L,1L,1L,1L,1D,0R,1R,1D,1R,1D,1R,1R,
          1R,1R,1R,1R,1R,1R,1R,1R,1U,1R,1U,1R,1U,1R,0U,1L,
          1L,1L,1L,1L,1L,1L,1U,1U,1U,1U,1U,1R,1U,1U,1L,1L,
          1D,1D,0D,0D,1L
```

```
9030 DATA OR,OR,OR,OU,1L,OD,1L,OD,1L,OD,1L,OD,1L,OD,1L,OD,
        1L,OD,1L,OD,1L,OD,9
```

Now assume that you saved the table under the name of ROW. We will write a program which will do the following:

```
load the shape table ROW
enter hi-res graphics and set parameters
loop across the screen
     xdraw figure 1
     delay time
     xdraw figure 1 (this is to erase it)
     change location
     xdraw figure 2
     delay time
     xdraw figure 2 (this erases figure 2)
     change location
endloop
```

Here is the program.

```
10 PRINT CHR$(4);"BLOAD ROW,A3328"
20 POKE 232,0: POKE 233,13
30 HOME:HGR: HCOLOR=3: SCALE=1: ROT=0
40 S=200 : REM S TIMES THE DELAY
50 FOR J=250 TO 10 STEP -8
60 XDRAW 1 AT J,40
70 FOR I=1 TO S: NEXT I
80 XDRAW 1 AT J,40
90 XDRAW 2 AT J-2,40
100 FOR I=1 TO S: NEXT I
110 XDRAW 2 AT J-2,40
120 NEXT J
```

You can use techniques based on a combination of shape tables and high resolution screens to create very interesting effects. Many very exciting graphics can be seen in some of the game programs on the market. If you have persevered through this introduction, you probably have a great appreciation for the time and effort which goes into developing these graphics.

11
Playing With Sounds

11.1 Making Sounds in BASIC

Early in Chapter Two we learned the Apple II could make a beeping sound by pressing down the CONTROL and the G key simultaneously. Try this to be sure you see how this is done. When you type this sequence, nothing appears on the screen. CONTROL-G is a non-printing character. It only generates the beep sound.

We can incorporate this beep sound into a program by telling the computer to print the CONTROL-G sequence. This will cause the bell to sound once. Because we cannot type in the CONTROL-G, we must enter it into our program using the CHR$ function. By looking at the table of ASCII codes given in Section 7.8, we see that the code for CONTROL-G is 7, so in the program we would ask the computer to print CHR$(7). To try this type in and run the following BASIC program.

```
10 PRINT CHR$(7)
```

This one-line program causes the bell to be beeped once. By putting it in a loop, we can have it beep repeatedly. Try this:

```
5 G$=CHR$(7)
10 FOR I=1 TO 10
20 PRINT G$
30 NEXT I
```

You can slow the bell down by using the SPEED command discussed in Section 5.5. Here is one example.

```
5 G$=CHR$(7): SPEED=50
10 FOR I=1 TO 10
20 PRINT G$
30 NEXT I
40 SPEED=255
```

TEST YOUR UNDERSTANDING 1

Try changing the speed in line 5 to see how this affects the beeping sound.

The Apple II Plus and IIe computers also have speakers. In order to access this speaker it is necessary to use a PEEK command as discussed in Chapter Eight. The memory address of the speaker is -16336. Thus, to access the speaker we must PEEK at this memory location. Doing so toggles the speaker and produces a clicking noise. Type and run this one-line program:

```
10 X=PEEK(-16336)
```

On running this program you should have heard a faint click. To make this more audible we can incorporate it in a loop. Run the program below:

```
5 S=-16336
10 FOR I=1 TO 10
20 X=PEEK(S)
30 NEXT I
```

TEST YOUR UNDERSTANDING 2

Try rerunning the above program changing the 10 in line 10 to 100. As you will see, this causes the sound to last longer.

You can also change the pitch a little by making the statements execute more quickly. This can be done by putting the entire FOR...NEXT loop on one line. To the above program add the following line:

```
40 FOR I=1 TO 100: X=PEEK(S): NEXT I
```

Note how the pitch changes. By playing with this device, you can exercise limited control on the pitch of the sound. You can use these sounds to alert users to the need for input data or to the existence of an error. In fact, if you type in an incorrect command on your Apple, it will beep to alert you. In the next section we will see how to use the speaker to make music!

11.2 Using Machine Language to Produce More Sounds

As we learn more and more about the BASIC language, we see that there are a lot of things we can make the computer do. There are some things which the computer can do which are just not available in BASIC. In order to get the computer to perform these kinds of tasks, it is necessary to write programs in the same "language" the machine understands. This language is called machine language. It has a more English-like counterpart called **assembly language**.

To really make use of the speaker's ability to generate tones, it is necessary to use instructions which cannot be written in BASIC. We are not going to talk about how you write programs in machine language or assembly language in this book, but we are going to poke a machine language subprogram into the computer's memory and then use that subprogram when

we wish to toggle the speaker. We can save this machine language subprogram as a binary file using the BSAVE command and load it back into memory using the BLOAD command.

Although discussion of programming in machine language is beyond the scope of this book, we will provide you with the necessary values to put into memory. When you wish to use the machine language subprogram, you will access it via a CALL statement.

The tone which the speaker emits can be varied in both duration and frequency. The frequency is related to the pitch of the tone you hear. The higher the frequency, the higher the pitch. The duration is related to the length of time you hear the tone. You will see this relation is not quite as direct as it might seem. Because the Apple II can execute some frequencies faster than others, the actual value assigned for duration must be larger for the higher frequencies. This will become clearer as we play with the sound program.

The Applesoft BASIC language allows for certain places in memory to be used for storing machine language programs. These memory locations normally start at location 768. Our machine language program will use locations 768 and 769 to store the values of the duration and frequency respectively. The machine language instructions will begin at location 770. When we are ready to execute this subprogram we will use the command

```
CALL 770
```

There are a couple of ways to enter a machine language subprogram into memory. Because it is not our purpose to dwell on the mechanics of machine language, we will use the technique of POKEing values into memory discussed in Chapter 8. We will poke the values beginning at location 770. Here is a BASIC program to place the data in memory and save it as a binary file called SOUNDMAKER.

```
10 FOR I=1 TO 26
20 READ X: POKE 769 + I,X
30 NEXT I
40 DATA 172,1,3,174,1,3,169,4,32,168,252,173,48,192,
     232,208,253,136,208,239,206,0,3,208,231,96
50 PRINT CHR$(4);"BSAVE SOUNDMAKER,A770,L26"
60 END
```

In order to actually use this machine language program, we need to load it into memory and assign values for the duration and frequency, then poke these values into memory and call the machine language subprogram. Type in the following program.

```
10 PRINT CHR$(4);"BLOAD SOUNDMAKER,A770"
20 INPUT "DURATION? ";D
30 INPUT "FREQUENCY? ";F
40 POKE 768,D
50 POKE 769,F
60 CALL 770
```

When you run this program, you will be asked for values of duration and frequency. These values must be integers between 1 and 255. Run the above program with the values given below to see the effect of changing values. You may wish to add the line

```
70 GOTO 20
```

to your program to save yourself from having to type in the word RUN each time. Stop the program by using the CONTROL-C sequence.

TEST YOUR UNDERSTANDING 3

Run the above program with the following values noting how the sound and duration changes.

duration	frequency
20	1
20	50
20	200
20	255
50	200
100	200
100	1
255	1

As you can see, it is very difficult to get low notes of long duration. By playing with the values of duration and frequency, you will be able to generate a wide range of tones.

We can begin to program the computer to write songs for us if we can approximate the musical scale. Here is one such approximation. Middle C is approximately frequency 87.

```
10 PRINT CHR$(4);"BLOAD SOUNDMAKER,A770"
50 GOTO 200
70 POKE 768,D
80 POKE 769,F
90 CALL 770
100 RETURN
200 REM SCALE APPROXIMATION
210 DIM A(31)
220 FOR I=1 TO 31: READ A(I): NEXT I
230 DATA 7,23,39,54,66,73,87,95,107,115,126,134,144,
     153,158,164,172,179,185,189,194,199,203,208,211,216,
     220,223,226,228,231
300 FOR I=1 TO 31
310    D=20:IF I=1 THEN D=70
320    GOSUB 70
330 NEXT I
```

The program uses the same duration for all frequencies except the lowest. You may wish to adjust these durations or frequencies to suit your ear.

We can use these scales to have the computer play tunes. Try this sequence:

```
10 PRINT CHR$(4);"BLOAD SOUNDMAKER,A770"
50 GOTO 200
70 POKE 768,D
80 POKE 769,F
90 CALL 770
100 RETURN
200 D=5: REM THREE BLIND MICE
220 READ F: IF F=0 THEN END
230 GOSUB 70
240 GOTO 220
250 DATA 126,107,87,87,126,107,87,87,153,134,126,126,
     153,134,126,126,0
```

11.3 Application 11-1: Row, Row, Row Your Boat

In this section we will use the scales developed in Section 11.2 to write a program to play "Row, Row, Row Your Boat." Then we will combine that with the animation we developed in Section 10.8. Our finished product will be a program which plays a song while the boat moves across the screen.

First, the program to play our song is a simple adaptation of the three blind mice routine. This time we will vary both the duration and the frequency.

```
10 PRINT CHR$(4);"BLOAD SOUNDMAKER,A770"
50 GOTO 200
70 POKE 768,D
80 POKE 769,F
90 CALL 770
100 RETURN
200 REM ROW ROW ROW YOUR BOAT
220 READ D,F: IF F=0 THEN END
230 GOSUB 70
240 GOTO 220
250 DATA 6,107,6,107,4,107,2,126,6,144
260 DATA 4,144,2,126,4,144,2,153,9,164
270 DATA 7,194,6,164,6,144,6,107
280 DATA 4,164,2,153,4,144,2,126,7,100,0,0
```

Now we can further embellish this program by attaching the animation we developed in Section 10.8. We would like to let the rowboat move across the screen as the music plays. Each successive note will allow the boat to move. If the note is of short duration, we will let a few notes play until the combined duration is at least six.

Here is the solution procedure.

```
load: load SOUNDMAKER into memory
      load ROW into memory (this is the rowing men shape
           table)
```

```
set parameters for high resolution graphics
draw and play:
     loop across screen
     xdraw first shape
     play first notes (until duration is at least 6)
     xdraw first shape
     xdraw second shape in new position
     play second notes
     xdraw second shape
endloop
```

Here is the complete program. Be sure you understand what each part of the program is doing. We will expand on these techniques in the next section.

```
10 REM LOAD SOUNDMAKER
20 PRINT CHR$(4):"BLOAD SOUNDMAKER, A770"
30 HOME
40 REM LOAD SHAPES
50 PRINT CHR$(4);"BLOAD ROW,A$D00"
60 POKE 232,0: POKE 233,13
70 HGR: HCOLOR=3: SCALE=1: ROT=1
80 FOR J=250 TO 10 STEP -8
90 XDRAW 1 AT J,40
100 READ D,F: IF F=0 THEN RESTORE: GOTO 100
108 GOSUB 7030
109 SD=SD+D: IF SD < 6 THEN 100
110 XDRAW 1 AT J,40
120 XDRAW 2 AT J-2,40
121 SD=0
126 READ D,F: IF F=0 THEN RESTORE: GOTO 126
127 GOSUB 7030
128 SD=SD+D: IF SD < 6 THEN 126
110 SD=0
120 XDRAW 2 AT J-2,40
130 NEXT J
200 END
1000 DATA 6,107,6,107,4,107,2,126,6,144
1010 DATA 4,144,2,126,4,144,2,153,9,164,7,194
1020 DATA 6,164,6,144,6,107,4,164,2,153,4,144,2,126,7,
       107,0,0
7020 RETURN
7030 REM SPEAKER ROUTINE
7040 POKE 768,D
7050 POKE 769,F
7060 CALL 770
7070 RETURN
```

11.4 Application 11-2: A Medley of Songs

Now that you know all about how to make your computer play songs, let's put some of the things we have learned about files, graphics, and sounds

together to let the computer play a medley of songs while displaying some graphics on the screen.

We will save the data statements corresponding to the various songs in text files. Then we can ask our main program to EXEC the appropriate file for each song.

Let's keep the data statements for the songs in lines 9000 through 9990 in each file and use flag values of 0,0 to signal the end of the song. For the "Row, Row, Row Your Boat" song we have the following data statements:

```
9000 DATA 6,107,6,107,4,107,2,126,6,144
9010 DATA 4,144,2,126,4,144,2,153,9,164
9020 DATA 7,194,6,164,6,144,6,107
9030 DATA 4,164,2,153,4,144,2,126,7,100,0,0
```

Using the program for creating EXEC files which we developed in Section 7.4, create a text file named ROW ROW ROW YOUR BOAT. Here is what that program will look like:

```
1 REM TYPE LINE NUMBERS IN LINE 5
2 INPUT "NAME OF FILE? ";A$
3 D$=CHR$(4): PRINT D$;"OPEN";A$
4 PRINT D$;"WRITE";A$
5 LIST 9000,9990
6 PRINT D$;"CLOSE";A$
7 END
9000 DATA 6,107,6,107,4,107,2,126,6,144
9010 DATA 4,144,2,126,4,144,2,153,9,164
9020 DATA 7,194,6,164,6,144,6,107
9030 DATA 4,164,2,153,4,144,2,126,7,100,0,0
```

When we run this program giving the file name as ROW ROW ROW YOUR BOAT, the program will create a text file containing these data statements. We can do this for each of the songs we want in our medley.

The main program will contain our machine language routine plus the instructions to EXEC each of the songs in turn and play them. Here is what it looks like:

```
10 PRINT CHR$(4);"BLOAD SOUNDMAKER,A770"
40 HOME: GOTO 200
70 POKE 768,D
80 POKE 769,F
90 CALL 770
100 RETURN
200 REM READ IN THE NUMBER OF SONGS AND THEIR NAMES
220 READ N: DIM S$(N)
230 FOR I=1 TO N: READ S$(I)
240 NEXT I
500 REM PLAY EACH SONG
510 FOR I=1 TO N
520 PRINT D$; "EXEC"; S$(I)
530 RESTORE
540 VTAB(22):PRINT S$(I)
550 READ D,F: IF F=0 THEN 570
```

```
560 GOSUB 70: GOTO 550
570 FOR J=1 TO 1000: NEXT J
580 NEXT I
600 END
9000 DATA 1
9010 DATA ROW ROW ROW YOUR BOAT
```

The first time through the program the machine language routines and the value of N, the number of songs, and the array S$ containing the song names are initialized via DATA statements starting in line 9000. As soon as the song file is EXECed, the DATA statements are replaced with data corresponding to the song. At that point we restore the data pointer and play the song.

Now let's think about adding some graphics. We allowed for the song title to be written at line 22 on the screen so that we could display this even with the graphics screen. We want to have some graphic effects which we can use with any song. Let's use just low resolution graphics and plot random squares on the screen in randomly chosen colors. Let's plot three squares for each note with each note in a randomly selected color.

To select random squares we will define a function to generate random integers between 0 and 39. For the choice of color we will generate random integers between 1 and 15 (this will exclude 0 which is black). Of course, you will not have the benefit of the color unless you have a color monitor, so you may wish to skip this part.

Here is the modified program. We have added the THREE BLIND MICE song so we have a medley of two. Remember that the program assumes that the correct DATA statements are in text files with the number of songs and their titles part of the original program.

```
10 PRINT CHR$(4);"BLOAD SOUNDMAKER,A770"
20 DEF FN X=INT(RND(1)*40)
30 DEF FN C=INT(RND(1)*15) + 1
40 HOME: GOTO 200
70 POKE 768,D
80 POKE 769,F
90 CALL 770
100 RETURN
200 REM READ IN THE NUMBER OF SONGS AND THEIR NAMES
220 READ N: DIM S$(N)
230 FOR I=1 TO N: READ S$(I)
240 NEXT I
500 REM PLAY EACH SONG
510 FOR I=1 TO N
520 PRINT D$; "EXEC"; S$(I)
530 RESTORE : GR
540 VTAB(22):PRINT S$(I)
550 READ D,F: IF F=0 THEN 570
560 COLOR=FNC
570 FOR J=1 TO 3
580 X(J)=FNX: Y(J)=FNX
590 PLOT X(J),Y(J)
600 NEXT J
```

```
610 GOSUB 70: GOTO 550
620 FOR J=1 TO 1000: NEXT J
630 NEXT I
640 END
9000 DATA 2
9010 DATA ROW ROW ROW YOUR BOAT
9020 DATA THREE BLIND MICE
```

Exercises

1. Compose some songs of your own and save them in text files. Add them to the medley of songs.

12
Software You Can Buy

As you probably have observed by now, it is not always easy to get a complex program up and running. Writing and debugging a complex program takes a considerable investment in time, wit, and dogged determination. In addition, building the most complicated programs takes a considerable amount of technical expertise in using the various features of the computer. Most people want to use their computer to simplify various everyday tasks but are not interested in building their major applications programs from the ground up. For these people there is a growing collection of programs available through computer stores and mail-order houses.

In this chapter we will introduce you to the general types of software available today and give you an idea of the features you may wish to look for when purchasing software. We will not make any specific recommendations because the best package for you is the one which meets your needs. This means you will have to decide which features are important to you and how often you will be using the package. Here are some questions that you should ask yourself as part of any software purchase.

Will the program run on my computer?

In order to run a particular program, you must have the proper operating system and an adequate amount of memory.

Programs are designed to run with a particular operating system. In this book we have discussed only Apple DOS. There are other operating systems. For example, there is the popular CP/M operating system. This operating system is available on most microcomputers, so there is an exceptionally large number of programs which run with CP/M. In any case, when buying a program, check to make sure that it will run under **your** operating system.

A description of a program will usually specify the amount of memory required. Obviously, you can't run a program requiring 64K of memory if you have only a 16K system!

Will the program do what I want it to?

When purchasing a program for a particular application, you will be faced with a wide range of choices. How can you properly choose among them? Frankly, this is a real problem.

As early as possible in the software selection process, you should define your own needs as exactly as you can. To determine the extent to which a given piece of software meets those needs, you must do a fair amount of digging. Your local computer store, reviews in computer journals, and software documentation are good sources of information.

Your local computer store is the first place to seek information. You will often be able to make a choice based on the information you get there. Many computer journals contain reviews on major pieces of software. Especially useful are reviews which compare several similar programs.

Another source of comparative information is the software documentation itself. You may often purchase the manual separately from the software. This is an expense, but it is often the only way you can be sure that the program will do exactly what you want it to do. After you have narrowed your program down to the top two contenders, why not purchase the manuals of each and inspect the two programs in more detail. It is much cheaper to invest in a manual you won't need than to spend several hundred dollars on a program which will not do everything you want it to!

What do I get for my software dollar?

When you purchase software, the minimum you get is the program on disk or cassette plus the applicable documentation. What else do you get?

1. Will the vendor accept phone calls seeking help in setting up and using the program?
2. What are the costs of updates to the software?
3. Will the vendor automatically notify you of major bugs in the software?
4. What is the quality of the documentation? Is it readily intelligible? Does it provide clear examples? Can it easily be used as a reference manual?
5. To what extent can you make required customizations of the software?
6. How much will it cost to replace defective copies of the program? Many software vendors currently supply you with only two copies of the program which cannot be copied further. You may have to pay to replace them if they are damaged. Some programs may be copied at will from the original disk.

You should not be intimidated by the discussion above. Most software purchases proceed quite smoothly. As with any other area of consumer affairs, it pays to be an informed purchaser. Hopefully, the suggestions above will help you spend your software dollars wisely.

The most common types of software fall into a few categories. These are: spread sheets, word processing, data base, accounting, graphics and educational packages (in addition to games, of course). We will look at each of these categories in turn and suggest some characteristics which may be important to you.

12.1 Spread Sheets—Visicalc et al

One of the most popular pieces of software ever created is the VISICALC* program. This is a financial planning program that can be used in hundreds

---

* VISICALC is a registered trademark of VISICORP, Inc.

of situations, both business and personal. Today there are several VISICALC look-alikes on the market, all designed to do essentially the same type of operation. The generic term for this type of package is a "spread sheet" program.

A spread sheet is nothing more than a table with columns and rows used as a worksheet for your calculations. The packages are designed so that entries in the table can be made to refer to other entries. Then if one value is changed, all other entries which refer to that value automatically reflect the change.

For example, if the first column of the table represented expenses throughout the month, and the last entry in that column was the sum of those expenses, anytime we changed the value of an expense or added a new expense, the sum would automatically be recalculated.

Let's illustrate a typical spread sheet application—a home budget. Suppose the following data describe the income and spending of the Smith family.

Income	
John	$1580 per month
Sally	$1145 per month
Interest	$280 in Jan., Apr., Jul.,Oct.
Bonus	$3500 in Dec.
Expenses	
Mortgage	$487.73 per month
Car payment	$187.50 per month
Utilities	$207.00 per month
Clothing	$200.00 per month
House repairs	$100.00 per month
Food	$640.00 per month
Entertainment	$120.00 per month
Fuel	$135.00 per month
Life insurance	$583.00 in Oct.
Vacation	$1,000.00 in Jul.
Health insurance	$285.00 in Apr. and Oct.
Pension plan	$400.00 per month

Everyone has probably compiled a list of this sort at one time or another. Once the list is compiled, there are a number of important questions to ask, such as

1. Is the Smiths' income sufficient to cover their expenses?
2. Is there any month when they will be short of cash?
3. How much is their annual savings (deficit)?

To answer such questions (especially the second one) the best approach is to compile a month-by-month description of income and expenses. Such a description could be written out on an accounting worksheet in the following fashion:

	Jan.	Feb.	Mar.	...	Dec.	Year
Income						
.						
.						
.						
Expenses						
.						
.						
.						
Total Income						
Total Expenses						
Income minus Expenses						
Cumulative Total						

We will leave it to you to fill in the above chart from the given data. If you do, you will discover that the Smiths are not making ends meet. There are a number of months where they have a deficit and they are running at a deficit for the entire year.

Being sensible people, the Smiths want to recast their budget so they are not spending more than they are earning or running a deficit in any month.

They begin to experiment with various cutbacks in expenditures. For each proposed cutback they must rework the above chart to determine if they have erased all the monthly deficits. It might take 15 or 20 tries before they arrive at a budget they can live with which is free of deficits. The calculations involved are quite burdensome if they are done by hand. They are exactly the sort of calculations which can be handled easily by one of the spread sheet packages.

The Smith's budget required a work sheet consisting of 14 columns and 20 rows. Other applications will require varying numbers of rows and columns. In order to make a wise purchase of such a software program, you should give some thought to the uses to which you might put the package. Be sure that the package you select can accommodate a sufficient number of rows and columns for your application.

Here is how a typical package might be used to solve the Smiths' problem. When the package is loaded into RAM (usually done by "booting" the software disk), a blank worksheet appears. The rows and columns are indicated by letters and/or numbers.

The first step is to enter the Smiths' data into the worksheet. The screen looks very much like the worksheet in Figure 12-1 (this is a typical VISICALC worksheet). You may only refer to a single worksheet position at a time, namely the position indicated by the cursor.

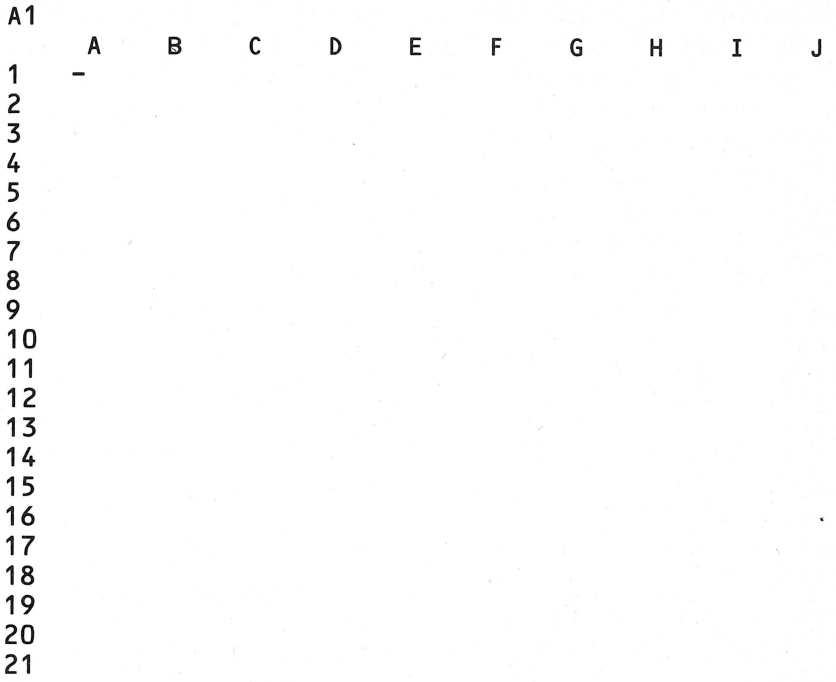


Figure 12-1. A VISICALC Worksheet.

In the worksheet of Figure 12-1, the currently indicated position is the first row, first column or position A1.

Our first task is to customize the worksheet to the data given in the Smiths' budget. Let's put the headings in the first row and first column. You move the cursor around the worksheet in a manner similar to that used for editing programs. Just position the cursor at the appropriate workspace position and type the desired entry. To correct typing mistakes use the Backspace key exactly as you would if you were typing a BASIC program. In our example we would move the cursor to position B1 and type "Jan" followed by **RETURN**. We would then move the cursor one position to the right and type "Feb" and so forth. We type the row labels beginning in the first column of row 2. The first line of the display always indicates the current position of the cursor. Let's now assume that the row and column headings are typed as in Figure 12-2.

In the example given the entries in the first column are truncated. This is a characteristic of VISICALC and is common to many of the packages. A certain column width is assumed and entries are truncated to fit that width. In some packages the heading will continue to type into other columns as needed. In any case, the packages allow you to adjust the column width to suit your needs. One feature you might want to look for is the ability to set column widths individually. In some packages, like VISICALC, each column must be the same width. In others you can pro-

```
A1
      A     B     C    D    E    F    G    H    I    J
1          Jan.  Feb. Mar. Apr. May  Jun. Jul. Aug.
Sep.
2 Income
3  John
4  Sally
5  Inter
6  Bonus
7
8 Expens
9  Mortg
10 Car p
11 Utili
12 Cloth
13 House
14 Food
15 Enter
16 Fuel
17 Life
18 Vacat
19 Healt
20 Pensi
21 Total
22 Total
23 Income
24 Cumula
```

Figure 12-2. VISICALC worksheet for the Smith budget.

vide for a wider column for long titles and shorter ones for numeric entries.

The penalty imposed by using wider columns is that fewer of them simultaneously fit on the screen. By moving the cursor we may bring the hidden columns into view.

We now wish to enter the budget data into the columns Jan.-Dec. Let's begin with the Jan. column. We must position the cursor at location B3. We could do this using a combination of one-space cursor moves, but most of the programs have some command to allow you to move directly to the desired cell. When the cursor is positioned at the desired cell, we can type John's income for January, 1580. We then move the cursor down one entry and type Sally's income for January, and so forth. In similar fashion, we enter all the data in the budget for the month of January.

Let's now calculate total income and total expenses for January. Total income for January is obtained by forming the sum of the entries B3-B6. This may be indicated on the worksheet by typing

```
+B3+B4+B5+B6
```

The leading + sign is necessary to notify VISICALC that B3 is not a heading but the location of an entry. Every spread sheet package allows

you to enter a formula instead of a number and has some means of letting the computer know that it is a formula. Once the formula is entered, the package will automatically insert the required sum at this location.

In addition to the straightforward expression given above, most packages also have built-in functions. They will usually have a SUM function to allow you to add consecutive entries in a column or a row simply by stating the starting and ending cell for the sum. For example, to calculate the *total expenses*, it is necessary to add up the entries in B9 through B20. In VISICALC we would position the cursor at position B22 and type

```
@SUM(B9.B20)
```

The entry in B22 will then be calculated as the sum of the entries in B9 through B20. Entry B23 *income minus expenses* is just

```
+B21-B22
```

Entry B24 is the same as B23, so we type

```
+B23
```

This completes the entry of the column corresponding to January.

To complete the second column it is not necessary to go through all the trouble we just went through for the month of January. All entries are the same except for the 280-dollar item for interest which must be changed to 0. Spread sheet packages have a feature which allows you to copy or replicate any sequence of entries. In VISICALC, to replicate the B column, we type

```
/R
```

followed by **RETURN**. At the top of the screen, you will see the prompt

```
Replicate:Source Range or RETURN
```

The computer is asking us what it is we wish to copy. Type

```
B2.B24
```

followed by RETURN to indicate that we wish to copy the contents of the B column. The program will respond with the prompt

```
Replicate: Target Range
```

Respond by typing

```
C2.M2
```

followed by **RETURN** to indicate that the source B2-B24 is to be replicated beginning in each of the locations C2, D2, E2, . . . , M2. (M is the thirteenth letter of the alphabet, corresponding to the 12 columns to be replicated.) This sequence of inputs will cause the numbers in each of the entries of column B to be copied across to the 11 other month columns of the worksheet. When a request is made to copy or replicate a formula, the computer needs to know how to treat the entries in the formula. It needs to know whether to copy the formula exactly (called an absolute

reference) or whether it is to be changed relative to its new position. If we copied the formula absolutely, then each column would contain sums from column B. Obviously, we want the sum formulas to be changed, so we obtain sums in the same column. Each package does this a little differently, so look for this feature and try to obtain a demonstration of how this is done on the package you are considering.

We can continue in this fashion, adding values and formulas as needed. When we have the worksheet in preliminary form, we will want to save it so we can use it over and over, either for planning purposes or for recording actual data. Spread sheet packages are usually designed to work in connection with some secondary storage system. You can save your worksheets either on disk or tape. For our VISICALC example let's assume we wanted to save the file under the name "budget." We would type in

```
/SS BUDGET
```

This saves the worksheet on the current drive.

To retrieve "budget" from the diskette on drive 1, type

```
/SL 1:BUDGET
```

Once the worksheet has been created, you may alter any entry in it; all the other entries will be recalculated according to the relationships specified.

In deciding upon a spread sheet package for your use, be sure to look at the following features and match them to your needs. In every case you will have to compromise among features and decide upon what you can afford to spend. This means that you must consider your needs beforehand.

Some things to consider when looking at spread sheet packages are:

- Cost
- Size of table (number of rows and columns) which can be accommodated.
- Format capabilities: can you have variable column widths?
  Can you fix the number of decimal places displayed?
  Can you display dollar signs?
- Function capabilities: How easy is it to enter a formula?
  What must be done to replicate a formula?
  What kind of built-in functions (like SUM) are available?
- Logical choice capabilities: Can you select different cell values depending on whether some condition is met (this is sometimes referred to as a table look up capability)?
- Help features: How good are the mnemonics used to help you remember commands?

12.2 Word Processing

In Chapter Nine we discussed the general concept of word processing. For anyone who seriously plans to use the computer for word processing, purchase of a package is mandatory. Although we cannot go into great detail, we will discuss some of the word processing you can expect your computer to accomplish using commercially available software.

Let's begin by describing some of the typical features of a word processing system you can run on your Apple II Computer. A word processing system is a computer program for creating, storing, and editing text.

At its most basic level, you use a word processing system like you would use a typewriter. Suppose you wish to prepare a document. You would turn on the computer and run the word processing program. The program first asks for the type of work you would like to perform. Possibilities include: type in a new document, edit an old document, save a document on diskette, or print a document. We would select the first option. Next we would describe various format parameters to the word processor: line width, number of characters per inch, number of lines per page, spacing between lines, and so forth.

We would then type the document exactly as we would on a typewriter, but there are several huge exceptions! First of all, we don't worry about carriage returns. The word processor takes care of forming lines. It accepts the text we type, decides how much can go on a line, forms the line, and displays it. Any text left over is automatically saved for the next line. The only function of the carriage return is to indicate a place where you definitely want a new line, such as at the end of a paragraph.

A second advantage of a word processor is in correcting errors. To correct an error we move the cursor to the site of the mistake, give a command to erase the erroneous letter(s) or word(s), and type in the replacements. Such action will generally destroy the structure of the lines (some lines may now be too long and others too short), but by using a simple command, it is possible to "reform" the lines according to the requested format.

Typically, a word processor has commands which enable you to scroll through the text of a document to look for a particular paragraph. Some word processors even allow you to mark certain points so you may turn to them without a visual search. When the document is finally typed to your satisfaction, you give the computer an instruction which saves a copy of it on diskette. At a future time you may recall the document and add to it at any point (even within the bodies of paragraphs!).

Typically, word processors have certain "block operations" which allow you to "mark" a block and then either delete it, copy it, or move it to another part of the document. You may also insert other documents into the current document. This is convenient in adding boiler plate, such as resumes, to your document. You may even use the block operations to alter boiler plate to fit the special needs of the current document.

You may construct your document in as many sessions as you wish. When your diskette finally contains the document as you want it, you finally give

the instruction to print. Your printer will now produce a finished, error-free copy of the document.

As if the above were not enough of an improvement over the conventional typewriter, the typical word processor can do even more. The features available depend on the word processor selected. Here are some of the goodies to look for.

Global Search and Replace. Suppose you wish to resubmit your document to another company, Acme Energetics. In your original document you included numerous references to the original company, Jet Energetics. A global search and replace feature allows you to instruct the computer to replace every occurrence of a particular phrase with another phrase. For example, we could replace every occurrence of "Jet Energetics" with "Acme Energetics". Global search and replace can be even more sophisticated. In some systems the word processor can be instructed to ask you whether or not to make each individual change. Another variation is to instruct the word processor to match any capitalization in the phrases replaced.

Centering. After typing a line you may center it using a simple command.

Boldface. You may print certain words in darker type.

Underscore. You may indicate underscoring portions of text.

Subscripts and Superscripts. You may indicate printing of subscripts (as in a_1) and superscripts (as in a^2). This is extremely useful for scientific typing. This function also depends upon the feature being available on your printer, so you will have to check both the hardware and the software in this case.

Justification. You may instruct the word processor to "justify" the right-hand margins of your text so that the text always ends exactly at the end of a line. This is possible only if you have a printer which is capable of spacing in increments smaller than the width of a single letter.

Spelling Correction. There now exist a number of spelling correction programs which compare words of your document against a dictionary (sizes range from 20,000 to 70,000 words). If the program doesn't find a match, it asks you if the word is spelled correctly and gives you an opportunity to add the word to the dictionary. In this way the output of a word processor can be proofread by computer.

Some things to consider when looking at word processing packages:

- Cost
- Cursor movement and scrolling: How easy is it to move around the pages of text? Can you move to the top of a page? to the top of the document?
- Format capabilities: Can you easily reset the format and reform paragraphs? Can you reformat the entire document at once if, for example, you wish to switch from single to double space? Can you add page headings or footings? Can you alter the page numbers? Does the system tell you when you have come to the end of a page?

- Block Functions: Can you move blocks of text around easily? Can you save sections of the document on disk? Can you merge two or more different documents together?
- Special Printing Features: Can you underscore, boldface, superscript, subscript? (Remember that some of these depend on your printer also.)
- Search features: Can you specify a string and have the computer find it for you? Can you automatically replace a string with another?
- Help features: How good are the mnemonics used to help you remember commands?

12.3 Data Base Packages

A data base is a collection of records which are logically related to each other. A data base package is a general program for manipulating such records of data which we want to use in many different types of applications.

The data bases created by the packages can be structured in many different ways. In the simplest form there are file management programs. These should probably not be called data base packages. They simply create and manipulate files much as we did in Chapter Seven. Other data bases are constructed so that the data in each record relates to and is linked to the data in the other records. This "linking" allows for some very sophisticated search and selection procedures, but the sophistication usually results in higher cost, more power or memory requirements, and greater time for processing.

Some things to consider when looking at data base packages are:

- Cost
- Size of the data base which can be accommodated. This may be expressed as a limit on the number of records, a limit on the size of any given record, or some combination of the two.
- Format and report capabilities: Can you change the format of a data base once it is in place, for example, if you suddenly want to add a field or change a field width? Can you select from the data in a record in order to output a specific report? Can you arrange the data on that report in any desired order?
- Search and sort capabilities: How many fields can you specify for searching and sorting? In other words, if you want to select records on the basis of part number, part name, or vendor code, you will need at least three *keys*. These are fields which you specify to be used in the selection process. How flexible is the selection process? Can you choose to print a report which contains only records which meet certain criteria? How do you have to specify that criteria? Can you select all values in a certain range? Can you use an IF . . . THEN capability for selection?
- Arithmetic capabilities: Can you total values in selected fields in a record? Can you total fields across several records? Can you find subtotals? If so, to how many levels?

- Command language: Can you write a program to use the capabilities of the database to do certain routine operations or does each step have to be entered each time?
- Help features: How good are the mnemonics used to help you remember commands?

12.4 Accounting

Most businesses of any size will eventually want to use the computer to keep their accounting records. There are many accounting packages on the market, and it is very important that you try to find one which accommodates your accounting peculiarities. Modification of such complicated packages is generally quite difficult. If you find a package which meets your needs except in some minor ways, it may be easier and more cost effective to modify your procedures rather than the package.

In general, you can purchase one integrated package or you can purchase individual modules. If you decide to purchase individual modules, you will want them to be integrated so that an entry made to one package can be directly entered into the other appropriate modules without additional data entry. Sometimes it is best to compromise on this feature in order to obtain a package which is suited to your needs.

The modules generally available are General Ledger, Accounts Receivable, Accounts Payable, and Payroll. If you are in a professional services business (for example, if you are an accountant or lawyer), you will probably also want a time accounting module. For certain professions you can find specially developed packages which account for the special needs of your business. It is wise to take your time and look around and, most of all, speak to others in your profession who are currently using some package.

Some things to consider when looking at accounting packages are:

- Cost
- Number of accounts, transactions and maximum dollar volume which can be accommodated. Sometimes, these interact, that is, the limit on the number of accounts may depend on the volume of transactions. Is it necessary to keep all data on one diskette? If so, does this provide enough storage space for your system?
- Formats allowed: Look at the way in which account numbers are specified. Are there enough digits allowed to match your current numbering system? Are alphabetic characters allowed?
- Report generation: How much flexibility do you have in formatting reports? Can the system generate the reports to which you are accustomed and which you need to operate your business?
- Ease of use: This is an important feature in any software system, but especially so in your accounting software. The easiest packages to use are generally menu driven, that is, they provide menus for option selection. Auditing features: If you have outside auditors, be sure that they find the package acceptable. How well does the package provide

audit trails? Can you enter transactions and then check them before actually posting them?

12.5 Graphics

Packages available to perform graphics routines can fall into several categories. The first are those which allow you to plot and chart data generated by other programs. Other packages specifically allow you to define shape tables and create pictures on the high resolution screens.

Some things to consider when looking at graphics packages are:

- Cost
- What printers are supported by the package? Several packages are designed to allow you to "dump" your high resolution screens onto a printer. This will probably be very important to you. Be sure that the package you purchase will support your printer. If it does not, you may not be able to use this feature to obtain hard copy. This graphics option is something you will also want to consider when purchasing a printer. See Chapter Thirteen for a discussion of this.
- What kind of graphics are provided by the package? If you are interested in a plotting and charting package, you will want to examine the ability of the package to put titles and legends on the graphs and charts. You may want a package which will support a color printer or one which will allow different patterns (vertical lines, horizontal lines, cross hatches) to be used to fill in different portions of the chart.

  If this package allows you to create shape tables and high resolution screens, what kind of shape tables are already provided? Some packages come with built-in shape tables. This is particularly helpful if the package contains shape tables with upper and lower case letters, numbers and special symbols. This will allow you to have text anywhere on the hi-res screen. Some packages even provide different character sets for typing in foreign languages.

  If you plan to use the package for design purposes, you may want three-dimensional graphics. This is not a standard feature so be sure to look for it specifically. Such packages are sometimes called Computer Aided Design (CAD) packages.

12.6 Educational Packages

Educational packages again fall into two categories. The first is the set of packages designed for persons who are in the education field and want to use the computer to develop computer aided lessons. These may allow you to actually develop instructional materials easily, to create multiple choice or true-false exams, or simply to keep student records.

The second category of educational programs contains programs actually designed to provide instruction and drill. These packages allow you to

improve memory skills, practice mathematics drills, or receive instruction in such topics as astronomy, optics, and chemistry.

Some things to consider when looking at education packages are:

- Cost
- Topics and level: Read all available documentation to be sure that the package will actually cover the material you are interested in learning and at an appropriate level.
- What kind of graphics are provided by the package? If the package does nothing more than provide screens full of written material, you could quickly become bored. A good educational package will support the text material with graphics, perhaps even animation. This is true whether it is a package designed to develop educational materials or simply deliver them.
- User control: When using an educational package, you may wish to skip over parts you already know or to repeat a section for review. Does the package allow you to do this easily? Can you select specific topics within the subject area? Are there quizzes which you can take to test your mastery of the material?
- Administrative features: If you are an educator using this package to deliver instructional materials or to administer tests, will the package keep track of student names and grades for you?

13
Expanding The System

13.1 Communicating with Other Computers

One of the advantages of owning a mircocomputer is that you now have the ability to communicate with other computers through your Apple II. In order to do so, you will need a device called a **modem** to effect the link. Modem stands for modulator-demodulator and is a device which takes the bit or digital signals your computer is using and converts them to wave or analog form used to transmit over phone lines. By hooking your computer to another over phone lines, you can transmit and receive data.

Modems vary substantially in terms of what they provide in addition to the modulation-demodulation. Some of them are acoustic couplers. This means that the phone receiver is physically placed in contact with the coupler. Others merely plug directly into a phone jack.

Some modems are programmable; you can write software programs (indeed, in some cases these are even supplied) which will allow the computer to dial in and sign onto specific computer systems without your physically having to enter the phone number each time. This can be a great time saver if you are using other systems often.

Finally, modems differ in the way that data is transmitted. Full discussion of this is beyond the scope of this text. In brief, the data can be transmitted in half or full duplex. This refers to the number of wires which are actually used for the transmission. A half duplex system means that transmission can occur in only one direction at a time. If you are sending information, the other computer cannot simultaneously be sending signals to your computer. In a full duplex system both computers can be sending at the same time. It is best to have a modem which can be switched into either mode because what you need will depend upon how the other computer is configured.

The speed of transmission is also important. Transmission speed is usually referred to as the **baud rate**. Usually, but not always, this refers to the number of bits per second (bps) which can be handled. The "standard" transmission speed is 300 bps, but speeds can be lower or higher. For a microcomputer, the typical range is from 300 to 1200 baud. Again, the speed required depends on the computer at the other end.

One of the nice things you can do if you have a modem is to tie into some of the standard news services. Some of these offer summaries of news events, local movie and theater listings, weather, and even daily horoscopes.

You are usually charged a fixed fee for joining plus a connect charge based on the length of time you are actually using the system with lower rates for off hours. Of course, you also have your phone charges. Computer clubs and stores often have bulletin board services you can tie into via phone where you can leave messages for other users and receive messages from them. Some software houses even provide for purchase and transmission of software packages via phone.

13.2 Printers

A printer is probably one of the extras you will want to add to your system almost immediately. This is particularly true if you plan to use the system for word processing, but even if you plan to do your own programming, you will find the printer helpful. It is much easier to debug a long program when you can see the entire thing spread out in front of you.

In selecting a printer there are three main qualities with which you should be concerned. These are cost, speed, and quality of print. Printers start at about $500 and go up from there. A critical determinant will be what you can afford. If you plan to use the printer a lot to produce long documents, the speed with which it can print is important. Speed is usually measured as characters per second (cps) which can be printed. Stated speeds vary starting at about 30 cps. This is the stated speed, not necessarily what you can actually expect to obtain. Actual output speed depends upon how the printing is done as well as the nature of the document you are producing.

Some printers are bi-directional. That means that they print while the carriage moves to the right, which is the normal mode of printing, as well as while the carriage is returning to the left. A bi-directional printer eliminates some of the delay which can be experienced due to carriage return delay.

Speed of printing can also be improved if the printer is logic seeking. This means that the carriage return will move only as far to the left as required to print the line. Without this feature the carriage return will always go to the extreme left before returning. Some delay can also be avoided with this feature.

Quality of print is largely determined by how the letters are formed. Basically, there are two categories of printers: formed character, also called letter quality, and dot matrix. A letter quality printer contains a wheel or a ball which has fully formed characters on it. An impression is made on the paper by having the wheel (often called a daisy wheel because of its appearance) or ball spin to locate the desired character and then impact the paper. It uses exactly the same principle as the normal typewriter.

A dot matrix printer, on the other hand, has a matrix of dots available to form every character. Different characters are formed by allowing specific dots to hit the paper. For example, if your printer has a 7 × 5 pattern, it means that the printer will use 7 rows of dots and 5 columns of dots to print every letter. Figure 13-1 shows such a 7 × 5 matrix with the appropriate dots darkened to show how a few letters might be printed. In general, the denser

the matrix used, the more the space between dots will be hidden and the more likely the print will be usable for letters.

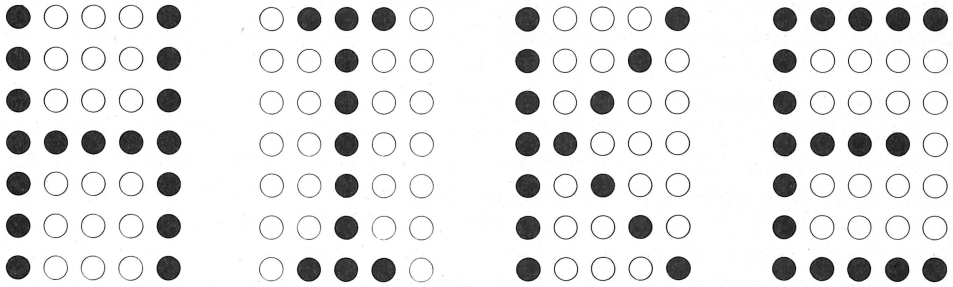


Figure 13-1. A typical dot matrix display.

There are a few other features to look for in selecting a printer. Here are some of the more significant ones.

Paper Width: Different printers will accommodate different paper widths. You should be sure that the printer you purchase will handle the size you frequently use. In particular, be sure it can handle a business size envelope. Some of the available printers cannot and this could be a real disadvantage for you. Be sure the paper width is adjustable so that you can type on envelopes, paper, and mailing labels with equal ease.

Tractor vs. Friction feed: Most printers will allow you to use paper which has tiny holes on either side. The paper is fed into the machine via tractors on either side of the printer. Frequently, the tractor holes on either side of the paper are perforated so that they can be torn off.

If you wish to use your own stationary, you will also need to have friction feed. In this mode the paper is held very tightly against the platen, and the holes on either side of the paper are not necessary.

Special Printing Features: If you need to use superscripts and subscripts, you will need a printer which supports this. It must be able to do a half line feed up or down. Similarly, check to see that the printer will support underlining and boldface if these options are important to you. Some printers also allow different character fonts. With a letter quality printer this is just a matter of changing the wheel or ball. With a dot matrix printer there may be special controls, either hardware or software, which allow you to print different fonts. If you need such things as Greek letters, italics, or mathematical symbols, be sure to check for this.

Graphics: Some printers will not allow you to print high resolution screens. Others have two different models, one which supports graphics and one which does not. In other printers the graphics capability may be an option which can be added later. The interface board you use in this case may also be important. If you plan to use graphics on the printer, be sure that both the printer and the interface board can handle it.

Buffers: Your printer probably has a print buffer area which can hold characters transmitted before they are printed. The size of this buffer will determine how quickly you will be able to regain control of your computer after

asking for something to be printed. It may also be important if you are having something printed from a remote location. If the sending computer is faster than your printer and you have no way to tell the sending computer to wait, you will begin to lose lines. A large buffer can eliminate much of this problem. (It can also be eliminated in other ways, but this is beyond the scope of this text.)

Software Support: The final point to consider is how well a given printer is supported by existing software. Many canned packages have built-in routines which allow you to print easily to certain printers. If you have a different make, you may have to write your own "driver" routine to drive the printer. While this is not an impossible task, having a printer already supported does make life easier.

13.3 Other Languages And Operating Systems

The Apple II computer uses the 6502 microprocessor and the Apple DOS operating system. Much of the software being developed today is based on the CP/M (control program for microcomputers) operating system. It is possible to add a board to your Apple II which will allow you to run programs written for the CP/M system. This board contains a Z80 microprocessing chip which means that, with the addition of this board, your computer will actually have two microprocessors and two possible operating systems.

In this case you will also have available another version of BASIC called MBASIC*. If you already know Applesoft, you will be able to learn MBASIC with little difficulty. Most of the commands are the same with the most substantial differences in the way that data files are handled.

You should be cautioned that even though the Apple II will be able to run CP/M programs, this does not mean that you can buy software for other computers and run it on your Apple II. You must still be sure the software was written for the Apple. This is primarily because of the way the disks are handled in different systems and is true for most microcomputers on the market today.

You can also add other programming languages to your Apple by adding different boards. The easiest of these is Integer BASIC in which all computing is done in integers or whole numbers. This language is good for working with strings or graphics. If you have added extra memory to your Apple II computer, this is automatically loaded when you boot the System Master diskette.

Other languages usually are more powerful than BASIC and may require some additional programming skills. These include the language Pascal as well as micro versions of popular mainframe languages like COBOL and FORTRAN. If you are interested in these options, speak to your microcomputer dealer. If you have successfully made your way through this book, you should be able to learn another language with much less difficulty. The fundamental ideas are similar, but the syntax rules change.

---

*MBASIC is a trademark of Microsoft Corporation.

13.4 Other Extras

We have only scratched the surface of all of the options which you could add to your Apple II. Among some of the others are:

80-column boards: These are boards which allow you to see 80 columns across the screen instead of just 40. It is a helpful addition if you are doing word processing. You will want to be sure that the software package supports an 80-column display; otherwise, it will do you no good.

If you have an Apple II Plus and purchase an 80-column board, there are some things you will have to do differently when writing BASIC programs. Consult the manual supplied with the board. The 80-column board which may be purchased for the Apple IIe supports all of the functions which we have described in this book.

Real-time Clock: You can also add a board which will keep track of the time. This will allow you to add features to your programs which are related to actual clock time.

Music Boards: We have discussed making music with your Apple II. If you are a professional composer, or if you simply would like to play around with making music, it is possible to buy software/hardware packages which allow you to compose music more easily and play it back using your stereo system. In some cases you can even buy piano-like keyboards which attach to the system, allowing you to enter the music by playing at a standard piano keyboard.

Speech Synthesizers: You can make your Apple II appear to emit sounds similar to human speech by adding a speech synthesizer. These boards emulate the frequencies used in speech to make your computer talk!

Paddles and Joysticks: This is a useful accessory if you plan to use your Apple II to play games. Paddles come in sets of two: one to control the up and down motion of the cursor and one to control the left/right movement. Each paddle has a button which is used to fire missiles or start games. On the Apple IIe the two Apple function keys can be used as if they were paddle buttons. The joystick is a single control which allows you to move the cursor around the screen by moving a single lever.

Plotters and Graphics Tablets: If you are planning to do a lot of plotting, you may find these useful additions to investigate. The graphics tablet is a large board on which a grid is printed. By moving a stylus about the tablet, the same image is transmitted to the computer.

A plotter is used for output of plots and charts. Here the computer controls the movement of a pen on paper. Use of a plotter allows you to draw smooth lines with the computer instead of the dot image which is typical of computer output.

Battery Packs: If you live in an area where there are frequent power outages, you may wish to add a battery pack. This will keep your computer from losing power when the current is off and allow time to save whatever you have entered into RAM. Indeed, you can even use a battery pack to take your computer outside and work in the sun!

Appendix—Using Cassette Tape for Secondary Storage

In this text, although we are assuming that the user has a disk drive available to save programs and data, the materials provided are applicable to those using cassette tape as a storage medium. In this case you will have to be careful not to record over any previously recorded material. This requires you to keep good records of where each file is saved on tape. This can be done by using the tape counter on the cassette recorder.

Saving a Program on Cassette

To save the FEET-TO-METERS program introduced in Chapter Two on cassette tape, we position the tape on a blank segment and push the PLAY and RECORD buttons of the cassette recorder simultaneously. Next, type the command

```
SAVE
```

The program will then be written onto the cassette. You should use the tape counter on the cassette recorder to identify the beginning of the program on the cassette. Note that the cassette recorder does not allow you to identify a program by a program name.

TEST YOUR UNDERSTANDING 1 (answer on page 309)

This is a BASIC program to add two numbers together.

```
10 INPUT "TYPE IN THE FIRST NUMBER TO ADD ";N1
20 INPUT "TYPE IN THE SECOND NUMBER TO ADD ";N2
30 S=N1 + N2
40 PRINT "THE SUM IS ";S
50 END
```

1. Type this program into RAM via the keyboard (remember to type in NEW to clear RAM first).
2. LIST the program to check for mistakes and correct these typing errors.
3. RUN the program a few times with different data values.

4. SAVE the program on tape.

Loading a Program from Cassette

To read the program back into RAM from cassette, rewind the tape and position it at the start of the program (use the tape counter to find the right spot), push the PLAY button on the cassette recorder, and type the command

```
LOAD
```

If you make changes in your program, you can save the new version as well as the old by saving it under a new name on diskette or by saving it on a blank section of tape on cassette.

Storing Arrays on Cassette

The Apple II Computer allows you to save an array on a cassette using the **STORE** instruction. To save A(I,J,K), we would use an instruction of the form

```
100 STORE A
```

Prior to running a program containing a **STORE** instruction, it is necessary to position the cassette to a blank portion of tape and to adjust the cassette recorder to record.

To recall the array from the cassette, use the instruction

```
200 RECALL A
```

In order to use a **RECALL** instruction, it is necessary to position the tape at the beginning of the area on which A was stored and to adjust the cassette recorder to play.

If you wish to do both **STORE** and **RECALL** in a single program, it will be necessary to program a **STOP** and a screen prompt into the program to tell you to reposition the tape and change the recorder from record to play.

ANSWER TO TEST YOUR UNDERSTANDING 1

After typing in your program and correcting the errors, type RUN. The output should look like this (this example uses input data values of 2 and 3):

```
TYPE IN THE FIRST NUMBER TO ADD 2
TYPE IN THE SECOND NUMBER TO ADD 3
THE SUM IS 5
```

To save the program on tape, position the tape to a point which has not yet been used to record programs (use your tape counter). Then press the PLAY and RECORD buttons simultaneously and type in

```
SAVE
```

Answers to Selected Exercises

CHAPTER 3

Section 3.1 (page 42)

1. a. VALID, string
   b. VALID, real
   c. VALID, real
   d. NOT VALID,cannot contain special symbol
   e. NOT VALID, must start with letter
   f. VALID, integer
   g. VALID, string
   h. VALID, real
2. 2.3E7 3. 1.7525E2 4. -2E8 5. 1.4E-4 6. -2.75E-10
7. 5.342E16 8. 159,000 9. -20,345,600
10. -.000000000007456 11. .00000000000000000239456
12. A can only assume numeric constants as values
13. Nothing
14. A$ can only assume string constants as values.
15. No line number. String constant not in quotes.
16. Nothing
17. A variable name must begin with a letter.

Section 3.2 (page 47)

1.
```
JONES          MARY ELIZABETH
```
2.
```
SILVER GOLD COPPER
  327   448  1052
```
3.
```
          FISH      MEAT
MON       1,245     2,348
TUE       248       3,459
```

Section 3.3 (page 51)

1.
```
10 X=57+23+48
20 PRINT X
30 END
```
2.
```
10 X=57.83 * (48.27-12.54)
20 PRINT X
30 END
```
3.
```
10 X=127.86/38
20 PRINT X
30 END
```
4.
```
10 X=365/.005 + 1.02^5
20 PRINT X
30 END
```
5. 10 6. 0 7. 50 8. 9 -7 18
9. JOHN JONES AGE 38 10. 22 57
11.
```
10 B=45 + 35 + 5
20 R=.80 * B
30 X=B - R
40 HOME
50 PRINT "AMOUNT OF BILL ";B
60 PRINT "MAJOR MEDICAL  ";R
70 PRINT "AMOUNT DUE     ";X
80 END
```

12.
```
10 A=2.3758
20 B=4.58321
30 C=58.11
40 X=A+B+C
50 Y=A*B*C
60 Z=A^2+B^2+C^2
70 PRINT X,Y,Z
80 END
```

Section 3.4 (page 59)

1.
```
10 PRINT "PROGRAM TO CONVERT METERS TO FEET"
20 INPUT "NUMBER OF METERS? ";MT
30 FT=3.2808 * MT
40 PRINT "EQUIVALENT # OF FEET=";FT
50 END
```
2.
```
10 PRINT "PROGRAM TO CONVERT LIQUID QUARTS TO LITERS"
20 INPUT "NUMBER OF LIQUID QUARTS? ";QT
30 LT=3.2808 * QT
40 PRINT "EQUIVALENT # OF LITERS=";LT
50 END
```
3.
```
10 PRINT "PROGRAM TO CONVERT LIQUID LITERS TO QUARTS"
20 INPUT "NUMBER OF LIQUID LITERS? ";LT
30 QT=0.9463 * LT
40 PRINT "EQUIVALENT # OF QUARTS=";QT
50 END
```
4.
```
10 PRINT "THE VALUE OF X IS ";5.378
20 END
```
5.
```
10 PRINT "THE VALUE OF X IS";TAB(22) 5.378
20 END
```
6.
```
10 PRINT TAB(20) "DATE";TAB(30) "3/18/81"
20 PRINT
30 PRINT "PAY TO THE ORDER OF WILDCATTERS, INC."
50 PRINT
60 PRINT "The Sum of";TAB(14) "*********$89,385.00"
```

Section 3.5 (page 69)

1.
```
10 DATA MARY JONES, 234-567, 150.00, 1000
15 DATA NOMORE,0,0,0
20 READ N$,CN$,BD, CL
30 IF N$="NOMORE" THEN END
40 PRINT "PLEASE TYPE IN CREDIT CARD NUMBER AND NEW CHARGES"
50 INPUT CC$,NC
60 IF CC$<>CN$ THEN PRINT "SORRY, WRONG CARD NUMBER":GOTO
     40
70 IF BD+NC < CL THEN PRINT "SORRY, CREDIT LIMIT
     EXCEEDED": GOTO 20
80 PRINT "CHARGE ACCEPTABLE": GOTO 20
```
2.
```
10 INPUT "TYPE VALUE OF NUMBER ";N
20 R=N^(1/2): S=N^2
30 PRINT ,"SQUARE"
40 PRINT "VALUE"," ROOT","SQUARE"
50 PRINT N,R,S
60 END
```
3.
```
10 N=1: REM START COUNTER FOR TEN PROBLEMS
20 INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
30 INPUT "WHAT IS THEIR PRODUCT";C
40 IF A * B=C THEN 200
```

```
50 PRINT "SORRY. THE CORRECT ANSWER IS",A*B
60 GO TO 500 : REM GO TO THE NEXT PROBLEM
200 PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
210 R=R+1 : REM INCREASE SCORE BY 1

500 IF N < 10 THEN N=N+1: GOTO 20
600 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
700 PRINT "TO TRY AGAIN, TYPE RUN"
800 END
```

4.
```
10 N=1: REM START COUNTER FOR TEN PROBLEMS
15 PRINT "CHOOSE OPERATION TO BE TESTED:"
16 PRINT "ADDITION(A), SUBTRACTION(S), OR MULTIPLICATION(M)"
17 INPUT A$
20 INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
30 IF A$="A" THEN 100
40 IF A$="S" THEN 200
50 IF A$="M" THEN 300
100 INPUT "WHAT IS THEIR SUM";C
110 IF A + B=C THEN 400
120 PRINT "SORRY. THE CORRECT ANSWER IS",A+B
130 GO TO 500 : REM GO TO THE NEXT PROBLEM
200 INPUT "WHAT IS THEIR DIFFERENCE";C
210 IF A - B=C THEN 400
220 PRINT "SORRY. THE CORRECT ANSWER IS",A-B
230 GO TO 500 : REM GO TO THE NEXT PROBLEM
300 INPUT "WHAT IS THEIR PRODUCT";C
310 IF A * B=C THEN 400
320 PRINT "SORRY. THE CORRECT ANSWER IS",A+B
330 GO TO 500 : REM GO TO THE NEXT PROBLEM
400 PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
410 LET R=R+1 : REM INCREASE SCORE BY 1
500 IF N < 10 THEN N=N+1: GOTO 15
600 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
700 PRINT "TO TRY AGAIN, TYPE RUN"
800 END
```

5.
```
10 PRINT "PLEASE INPUT THE COEFFICIENTS A, B, AND C";
20 INPUT A,B,C
30 Y=B^2 - 4*A*C
40 IF Y < 0 THEN 300
50 X1=(B + Y^(1/2))/(2*A)
60 X2=(B - Y^(1/2))/(2*A)
70 PRINT "THE ROOTS ARE: ";X1,;" AND ";X2
80 END
300 T1=B/(2*A): T2=(-Y)^(1/2)/(2*A)/(2*A)
310 PRINT "THE ROOTS ARE IMAGINARY"
320 PRINT "THE REAL PART IS ";T1
330 PRINT "THE IMAGINARY PART IS ";T2;"I"
340 END
```

Section 3.8 (page 83)

1.
```
10 INPUT X
20 XI=X - INT(X)
30 PRINT "THE FRACTIONAL PART OF ";X;" IS "; XI
40 END
```

2. a.
```
10 I=1
20 X=100*RND(1)
30 PRINT X
```

b.
```
10 I=1
20 X=INT(50 * RND(1)+1)
30 PRINT X
```

```
40 X=X+1
50 IF X < 11 THEN GOTO
  20
60 END
```

```
40 X=X+1
50 IF X < 11 THEN GOTO 20
60 END
```

3. a. `10 X=EXP(1.54)` b. `10 X=EXP(-2.376)`
   c. `10 X=LOG(58)` d. `10 X=LOG(9.75E-5)`
   e. `10 X=SIN(3.7)` f. `10 X=COS(.017453*45)`
   g. `10 X=ATN(1)` h. `10 X=TAN(.682)`

4.
```
10 X=-1.0
20 PRINT X, EXP(X)
30 X=X + .1
40 IF X <=1.0 THEN GOTO 20
50 END
```

5.
```
10 DATA 1.7, 3.1, 5.9, 7.8, 8.4, 10.1, -1
20 READ X
30 IF X < 0 THEN END
40 Y=3*X^(1/4)*LOG(5*X) + EXP(-1.8*X)*TAN(X)
50 PRINT X,Y
60 GOTO 20
```

6. In this problem the values of SIN(X) will vary between -1 and $+1$, therefore the TAB setting must allow for this. The fundamental program without any attempt at marking the axes is:

```
10 X=0
20 Y=SIN(X)
30 PRINT TAB(30 + 10*Y);"0"
40 X=X+.4
50 IF X <=25 THEN GOTO 20
60 END
```

7. Drop line 20 from the program given in the answers to Exercise 4, Section 3.5. Substitute the following lines:

```
20 A=INT(100 * RND(1))
21 B=INT(100 * RND(1))
22 PRINT "THE TWO NUMBERS ARE:"
23 PRINT: PRINT
24 PRINT A,B
```

CHAPTER 5

Section 5.1 (page 101)

1.
```
10 S=0
20 FOR J=1 TO 25
30 S=S+J^2
40 NEXT J
50 PRINT S
60 END
```

2.
```
10 S=0
20 FOR J=0 TO 10
30 S=S+(1/2)^J
40 NEXT J
50 PRINT S
60 END
```

3.
```
10 S=0
20 FOR J=1 TO 10
30 S=S+J^3
40 NEXT J
50 PRINT S
60 END
```

4.
```
10 S=0
20 FOR J=1 TO 100
30 S=S+1/J
40 NEXT J
50 PRINT S
60 END
```

5.
```
10 PRINT "N","N^2","N^3"
20 FOR J=1 TO 12
30 PRINT J,J^2,J^3
40 NEXT J
50 END
```

6.
```
10 PRINT "MONTH","INTEREST","BALANCE"
20 B=4000,P=125.33
30 FOR J=1 TO 12
40  I=.01*B:REM I=THE INTEREST FOR MONTH
50  R=P-I:REM R=REDUCTION IN BALANCE FOR MONTH
60  B=B-R:REM  NEW BALANCE
```

```
70  PRINT J,I,B
80 NEXT J
90 END
```

7.
```
10 PRINT "END OF YEAR", "BALANCE"
20 B=1000
30 FOR J=1 TO 15
40 B=B +.10*B :REM ADD DEPOSIT AND INTEREST
50 PRINT J,B
60 NEXT J
70 END
```

8.
```
10 S=3.5E7: P=5.54E6
20 PRINT "END OF YEAR", "SALES", "PROFITS"
30 FOR J=1 TO 3
40  S=1.2*S: P=1.3*P
50  PRINT J,S,P
60 NEXT J
70 END
```

Section 5.3 (page 117)

1. `DIM A(5)` 2. `DIM A(2,3)` 3. `DIM A$(3)` 4. `DIM A(3)`
5. `DIM A$(4),B(4)`
6. a. `A(1)=2, A(2)=4, A(3)=6, A(4)=8, A(5)=10, A(6)=12, A(7)=14, A(8)=16, A(9)=18, A(10)=20`
   b. `A(0)=1.1, A(1)=3.3, A(2)=5.5, A(3)=7.7, B(0)=2.2, B(1)=4.4, B(2)=6.6, B(3)=8.8`
   c. `A(0)=1,B(0)=2,A(1)=3,B(1)=4,A(2)=1,B(2)=2,A(3)=3,B(3)=4`
   d. `A(1,1)=1, A(1,2)=2, A(1,3)=3, A(1,4)=4, A(2,1)=5, A(2,2)=6, A(2,3)=7, A(2,4)=8, A(3,1)=9, A(3,2)=10, A(3,3)=11, A(3,4)=12`
7.
```
10 DIM A$(3),B(3,2),C$(2)
20 PRINT ,"RECEIPTS"
30 C$(1)="STORE #1",C$(2)="STORE #2"
40 A$(1)="1/1-1/10",A$(2)="1/11-1/20",A$(3)="1/21-1/31"
50 B(1,1)=57385.48,B(1,2)=89485.45
60 B(2,1)=39485.98,B(2,2)=76485.49
70 B(3,1)=45467.21,B(3,2)=71494.25
100 PRINT ,C$(1),C$(2)
200 FOR J=1 TO 3
220 PRINT A$(J),B(J,1),B(J,2)
230 NEXT J
300 END
```

8. Add the instructions:
```
5 DIM D(2)
240 FOR J=1 TO 2
250 D(J)=B(1,J)+B(2,J)+B(3,J)
260 NEXT J
270 PRINT "TOTALS",D(1),D(2)
```

9. Move the END to 400 and add the following instructions.
```
6 DIM E(3)
300 FOR J=1 TO 3
310 E(J)=B(J,1)+B(J,2)
320 NEXT J
330 PRINT
340 PRINT  "PERIOD", "TOTAL SALES"
350 FOR J=1 TO 3
360 PRINT A$(J) , E(J)
370 NEXT J
400 END
```

10.
```
10 DIM A$(4), B$(5), C(5,4)
20 A$(1)="STORE #1":A$(2)="STORE #2": A$(3)="STORE #3"
21 A$(4)="STORE #4"
30 B$(1)="REFRIG.":B$(2)="STOVE":B$(3)="AIR COND."
40 B$(4)="VACUUM": B$(5)="DISPOSAL"
50 PRINT "INPUT THE CURRENT INVENTORY"
60 FOR J=1 TO 4
70 PRINT A$(J)
80 PRINT
90 FOR I=1 TO 5
100 PRINT B$(I)
110 INPUT C(I,J)
120 NEXT I
130 NEXT J
200 REM REST OF PROGRAM IS FOR INVENTORY UPDATE
210 PRINT "CHOOSE ONE OF THE FOLLOWING"
220 PRINT "RECORD SHIPMENTS(R) "
230 PRINT "DISPLAY CURRENT INVENTORY(D)"
240 INPUT "TYPE R OR D";D$
250 IF D$="R" THEN 300
260 IF D$="D" THEN 600
270 HOME:GOTO 200
300 HOME
310 PRINT "RECORD SHIPMENT"
320 INPUT "TYPE STORE#(1-4)";J
330 PRINT "ITEM SHIPPED"
340 PRINT "REFRIG=1,STOVE=2,AIR COND.=3,VACUUM=4,DISPOSAL=5"
350 INPUT I
360 INPUT "NUMBER SHIPPED";S
370 B(I,J)=B(I,J)-S
380 GOTO 200
600 HOME
620 FOR J=1 TO 5
630 FOR I=1 TO 4
640 PRINT A$(I),B(I,J)
650 NEXT I
660 NEXT J
670 GOTO 200
1000 END
```
Note that this program is really an infinite loop. For this type of program this is a good idea. You don't want to accidentally end the program thereby erasing the current inventory figures! End this program using the CONTROL-C key combination.

11. Add the following instructions:
```
132 IF C(J) > A(J) THEN 135
133 GOTO 140
135 PRINT "BET INVALID:NOT ENOUGH CHIPS PURCHASED"
137 C(J)=0
139 GOTO 120
```

12. Change line 132 in Exercise 11 to and add line 133:
```
132 IF C(J) > A(J)+100 THEN 135
133 GOTO 140
```

13. Set F(J) equal to the Federal withholding for employee J, N(J) = the net pay, and add the following lines.
```
280 PRINT "EMPLOYEE", "WITHHOLDING","NET PAY"
290 FOR J=1 TO 5
300 IF D(J) <=200 THEN F(J)=0
310 IF D(J) <=210 THEN F(J)=29.10
320 IF D(J) <=220 THEN F(J)=31.20
```

```
330 IF D(J) <=230 THEN F(J)=33.80
340 IF D(J) <=240 THEN F(J)=36.40
350 IF D(J) <=250 THEN F(J)=39.00
360 IF D(J) <=260 THEN F(J)=41.60
370 IF D(J) <=270 THEN F(J)=44.20
380 IF D(J) <=280 THEN F(J)=46.80
390 IF D(J) <=290 THEN F(J)=49.40
400 IF D(J) <=300 THEN F(J)=52.10
410 IF D(J) <=310 THEN F(J)=55.10
420 IF D(J) <=320 THEN F(J)=58.10
430 IF D(J) <=330 THEN F(J)=61.10
440 IF D(J) <=340 THEN F(J)=64.10
450 IF D(J) <=350 THEN F(J)=67.10
500 N(J)=D(J)-E(J)-F(J)
600 PRINT B$(J),F(J),N(J)
700 NEXT J
```

14.
```
5 DIM A(25)
10 DATA 10,10,9,9,8,11,15,18,20,25,31,35,38,39,40,40,42,38
20 DATA 33,27,22,18,15,12
30 FOR J=0 TO 23
40 READ A(J)
50 S=S+A(J)
60 NEXT J
70 PRINT "AVERAGE 24 HOUR TEMP.", S/24
100 PRINT "TO FIND THE TEMPERATURE AT ANY PARTICULAR HOUR"
110 PRINT "TYPE THE HOUR IN 24-HOUR NOTATION: 0-12=AM"
120 PRINT "13-24=PM"
130 PRINT "TO END THE PROGRAM, TYPE 25"
140 INPUT "DESIRED HOUR";A
150 IF A=25 THEN 200
160 PRINT "THE QUERIED TEMPERATURE WAS ";A(J);" DEGREES"
170 GOTO 100
200 END
```

15. Put your names in a series of DATA statements located in lines 1000-1010.

```
 5 DIM A$(10)
10 FOR J=1 TO 10
20 READ A$(J)
30 NEXT J
40 FOR J=1 TO 4
50 PRINT A$(10*RND(1)+1)
60 NEXT J
70 END
```

Section 5.5 (page 127)

1. Assuming we wish to print the line at row 10 on the screen, here is the solution procedure:

```
write out the initial statement at row 10
set cursor to first 3 column field (column 25)
     get first three digits
set cursor to next 2 column field (column 29)
     get next two digits
set cursor to next 4 column field (column 32)
     get final 4 digits
```

Here is the program:

```
10 HOME: VTAB(10)
20 PRINT "SOCIAL SECURITY NUMBER? ___-__-____"
30 VTAB(10):HTAB(25):J=3
```

```
40 FOR I=1 TO J
50 GET A$:PRINT A$;
60 NEXT I
70 IF J=2 THEN HTAB(32): J=4: GOTO 40
80 IF J=3 THEN HTAB(29): J=2: GOTO 40
90 END
```

2. Drop line 20 from the program of exercise 1 and substitute the following:

```
20 PRINT "SOCIAL SECURITY NUMBER? ";
21 FLASH: PRINT "____";: NORMAL: PRINT "-";
22 FLASH: PRINT "__";: NORMAL: PRINT "-";
23 FLASH: PRINT "____": NORMAL
```

Section 5.7 (page 145)

1. Here are the errors:
   TYPE MISMATCH in line 10: "0" should be 0
   line 30: J(2 should be J^2
   line 40: NEXT K should be NEXT J
   line 80: NXT T should be NEXT J
   line 90: should be deleted
   line 100: ST should be S*T
   line 110 quotes around "THE ANSWER IS"
2. line 30 should read: PRINT "THE FIRST N EQUALS",N
   Need line 40: GOTO 200

CHAPTER 6

Section 6.1 (page 150)

1. `10 DEF FNA(X)=X^2 - 5*X`
2. `10 DEF FNA(X)=1/X - 3*X`
3. `10 DEF FNA(X)=5*EXP(-2*X)`
4. `10 DEF FNA(X)=X*LOG(X/2)`
5. `10 DEF FNA(X)=TAN(X)/X`
6. `10 DEF FNA(X)=COS(2*X) + 1`
7.
```
10 DEF FNA(X)=5*EXP(-2*X)
20 FOR X=0 TO 10 STEP .5
30 PRINT X, FNA(X)
40 NEXT X
50 END
```

Section 6.2 (page 158)

1.
```
10 FOR J=.1 TO .5 STEP .1
20 GOSUB 100
30 PRINT X
40 NEXT J
50 END
100 X=5*J^2 - 3*J
110 RETURN
```
2. `1000 C(J)=100*(B(J)-A(J))/A(J)`
3.
```
2000 M=C(1)
2010 FOR J=2 TO 6
2020 IF M > C(J) THEN M=C(J)
2030 NEXT J
2040 K=1
2050 IF M=C(K) THEN 2100
2055 GOTO 2060
2060 K=K+1
2070 GOTO 2050
2100 RETURN
```

5. Let D(J)=4 mean that J bets on first 12, D(J)=5 that J bets on second 12, D(J)=6 that J bets on third 12. In all such bets B(J) will be 0. Corresponding to the new values of D(J), there will be three new subroutines, starting in lines 4000, 5000, and 6000, respectively. Modify lines 121 to 125 as follows:

```
121 PRINT "BET TYPE:1=NUMBER BET,2=EVEN,3=ODD,4=1st 12"
122 PRINT "5=2nd 12, 6=3rd 12"
123 INPUT "BET TYPE(1-6)";D(J)
124 IF D(J)>1 THEN 125
125 GOTO 130
126 INPUT "AMOUNT";C(J)
127 GOTO 180
```

Replace lines 320-330 by:

```
320 ON D(J) GOSUB 1000,2000,3000,4000,5000,6000
```

Finally, here are the three new subroutines.

```
4000 FOR K=1 TO 12
4010 IF X=K THEN 4100
4015 GOTO 4020
4020 PRINT "PLAYER";J;"LOSES"
4030 A(J)=A(J)-C(J)
4050 RETURN
4100 PRINT "PLAYER";J;"WINS";2*C(J);"DOLLARS"
4110 A(J)=A(J)+2*C(J)
4120 RETURN
```

The subroutines in 5000 and 6000 are identical, except for the lines:

```
5000 FOR K=13 TO 24
6000 FOR K=25 TO 36
```

6. The statements to obtain the digits will be placed in a subroutine starting at line 500. The program is:

```
10 HOME: VTAB(10)
20 PRINT "SOCIAL SECURITY NUMBER? ___-__-____"
30 VTAB(10): HTAB(25): J=3: GOSUB 500
40 HTAB(29): J=2: GOSUB 500
50 HTAB(32): J=4: GOSUB 500
90 END
500 FOR I=1 TO J
510 GET A$:PRINT A$;
520 NEXT I
530 RETURN
```

CHAPTER 7

Section 7.2 (page 173)

1.
```
10 DATA 5.7,-11.4,123,485,49
20 FOR J=1 TO 5
30 READ A(J)
40 NEXT J
50 PRINT CHR$(4);"OPEN NUMBERS"
55 PRINT CHR$(4);"WRITE NUMBERS"
60 FOR J=1 TO 5
70 PRINT A(J)
80 NEXT J
90 PRINT CHR$(4);"CLOSE NUMBERS"
100 END
```

2.
```
10 PRINT CHR$(4);"OPEN NUMBERS"
15 PRINT CHR$(4);"READ NUMBERS"
20 FOR J=1 TO 5
30 INPUT A(J)
40 NEXT J
```

```
50 PRINT CHR$(4); "CLOSE NUMBERS"
60 FOR J=1 TO 5
70 PRINT A(J)
80 NEXT J
100 END
```

3.
```
5 PRINT CHR$(4); "OPEN NUMBERS"
10 PRINT CHR$(4);"APPEND NUMBERS"
15 PRINT CHR$(4);"WRITE NUMBERS"
20 DATA 5, 78, 4.79, -1.27
30 FOR J=1 TO 4
40 PRINT A
50 NEXT J
60 PRINT CHR$(4);"CLOSE NUMBERS"
70 END
```

4.
```
5 PRINT CHR$(4); "OPEN NUMBERS"
10 PRINT CHR$(4); "READ NUMBERS"
20 FOR J=1 TO 9
30 INPUT A(J)
40 NEXT J
50 PRINT CHR$(4); "CLOSE NUMBERS"
60 FOR J=1 TO 9
70 PRINT A(J)
80 NEXT J
100 END
```

5.
```
5 J=0
10 DIM A(100),B$(100),C$(100),D(100),E$(100)
20 PRINT "TYPE CHECK DATA ITEMS REQUESTED."
30 PRINT "FOLLOW EACH ITEM BY A CARRIAGE RETURN."
40 J=J+1
50 INPUT "CHECK #";A(J)
60 INPUT "DATE";B$(J)
70 INPUT "PAYEE";C$(J)
80 INPUT "AMOUNT(NO $)";D(J)
90 INPUT "EXPLANATION";E$(J)
100 INPUT "ANOTHER CHECK(Y/N)";F$
120 HOME
130 IF F$="Y" THEN 20
200 PRINT CHR$(4); "OPEN CHECKS"
210 PRINT CHR$(4); "WRITE CHECKS"
220 PRINT J
230 FOR M=1 TO J
240 PRINT A(M),B$(M),C$(M),D(M),E$(M)
250 NEXT M
260 PRINT CHR$(4); "CLOSE CHECKS"
1000 END
```

6.
```
10 PRINT CHR$(4); "OPEN CHECKS"
20 PRINT CHR$(4); "READ CHECKS"
30 INPUT J
40 FOR M=1 TO J
50 INPUT A(M),B$(M),C$(M),D(M),E$(M)
60 NEXT M
100 PRINT CHR$(4):"CLOSE CHECKS"
110 S=0
120 FOR M=1 TO J
130 S=S+D(M)
140 NEXT M
150 PRINT "TOTAL OF CHECKS IS",S
1000 END
```

Section 7.5 (page 187)

1.
```
10 DATA 5.7,-11.4,123,485,49
20 PRINT CHR$(4);"OPEN NUMBERS,L10"
30 FOR J=5 TO 1 STEP -1
30 READ A(J)
40 PRINT CHR$(4);"WRITE NUMBERS,R";J
50 PRINT A(J)
60 NEXT J
70 PRINT CHR$(4);"CLOSE NUMBERS"
100 END
```
2.
```
10 INPUT "WHAT RECORD NUMBER? ";N
20 PRINT CHR$(4);"OPEN NUMBERS,L10"
30 PRINT CHR$(4);"READ NUMBERS,R";N
40 INPUT A
50 PRINT CHR$(4); "CLOSE NUMBERS"
70 PRINT "THE VALUE FOR RECORD ";N;" IS ";A
100 END
```
3.
```
10 PRINT CHR$(4); "OPEN NUMBERS,L10"
20 PRINT CHR$(4);"WRITE NUMBERS,R3"
40 PRINT 428
50 PRINT CHR$(4);"CLOSE NUMBERS"
100 END
```
4. We will begin by inputting all of the information into arrays. Then we will write this information into the CHECKS file allowing 9 bytes for the date, 20 for the payee, 10 for the amount, and 30 for the explanation for a total record length of 69.
```
5 J=0
10 DIM A(100),B$(100),C$(100),D(100),E$(100)
20 PRINT "TYPE CHECK DATA ITEMS REQUESTED."
30 PRINT "FOLLOW EACH ITEM BY A CARRIAGE RETURN."
40 J=J+1
50 INPUT "CHECK #";A(J)
60 INPUT "DATE";B$(J)
70 INPUT "PAYEE";C$(J)
80 INPUT "AMOUNT(NO $)";D(J)
90 INPUT "EXPLANATION";E$(J)
100 INPUT "ANOTHER CHECK(Y/N)";F$
120 HOME
130 IF F$="Y" THEN 20
200 PRINT CHR$(4); "OPEN CHECKS, L69"
210 FOR M=1 TO J
220 N=A(M) - 99
230 PRINT CHR$(4); "WRITE CHECKS,R";N
240 PRINT B$(M): PRINT C$(M): PRINT D(M): PRINT E$(M)
240 NEXT M
250 PRINT CHR$(4); "WRITE CHECKS,RO"
260 PRINT J
270 PRINT CHR$(4); "CLOSE CHECKS"
1000 END
```
5.
```
10 INPUT "INFORMATION ABOUT WHICH CHECK NUMBER? ";C
20 PRINT CHR$(4); "OPEN CHECKS,L69"
30 N=C - 99
40 PRINT CHR$(4); "READ CHECKS,RO"
50 INPUT J
60 IF N > C THEN PRINT "SORRY, INFORMATION ON THE FIRST
     ";J; "CHECKS ONLY": PRINT CHR$(4);"CLOSE CHECKS":
     GOTO 10
70 PRINT CHR$(4); "READ CHECKS,R";N
```

```
80 INPUT B$,C$,D,E$
90 PRINT CHR$(4);"CLOSE CHECKS"
100 PRINT "INFORMATION FOR CHECK # ";C
110 PRINT "DATE: ";B$
120 PRINT "PAYEE: ";C$
130 PRINT "AMOUNT: $";D
140 PRINT "EXPLANATION: ";E$
1000 END
```

CHAPTER 9

Section 9.1 (page 225)

3.
```
10 A$="15+48+97=160"
20 B$(1)=LEFT$(A$,2)
30 B$(2)=MID$(A$,4,2)
40 B$(3)=MID$(A$,7,2)
50 B$(4)=RIGHT$(A$,3)
60 FOR J=1 TO 4
70 B(J)=VAL(B$(J))
80 NEXT J
90 FOR J=1 TO 3
100 PRINT SPC(3-LEN(STR$(B(J)))); B(J)
110 NEXT J
120 PRINT "---"
130 PRINT SPC(3-LEN(STR$(B(J)))); B(4)
140 END
```

4.
```
10 A$="$6718.49": B$="$4801.96"
20 A1$=RIGHT$(A$,7):B1$=RIGHT$(B$,7)
30 A2=VAL(A1$): B2=VAL(B1$)
40 C=A2 + B2
50 PRINT "$";SPC(8-LEN(A1$);A2
60 PRINT "$";SPC(8-LEN(B1$);B2
70 PRINT "----------"
80 PRINT "$";SPC(8-LEN(STR$(C))); C
90 END
```

CHAPTER 10

Section 10.1 (page 244)

1.
```
5 GR:COLOR=1
10 HLIN 0,39 AT 18
```

2.
```
5 GR: COLOR=1
10 VLIN 0,39 AT 17
```

3.
```
5 GR:COLOR=1
10 HLIN 0,39 AT 19
20 VLIN 0,39 AT 19
```

4.
```
5 GR:COLOR=1
10 HLIN 0,39 AT 13
20 HLIN 0,39 AT 26
30 VLIN 0,39 AT 13
40 VLIN 0,39 AT 26
```

5.
```
5 GR: COLOR=1
10 VLIN 1,24 AT 30
20 VLIN 1,24 AT 31
```

6.
```
5 GR: COLOR=1
10 FOR J=0 TO 39
20 PLOT J,J
30 NEXT J
40 END
```

7.
```
5 GR: COLOR=1
10 FOR J=1 TO 40
20 PLOT J,12
30 NEXT J
40 FOR K=1 TO 3
50 PLOT 10*K,13
60 PLOT 10*K,11
70 NEXT K
80 END
```
8.
```
5 GR: COLOR=1
10 FOR J=1 TO 40
20 PLOT 19,J
30 NEXT J
40 FOR J=0 TO 4
50 PLOT 18,8*J
55 PLOT 20,8*J
60 NEXT J
70 END
```

Section 10.2 (page 252)

1. Add the following lines, and change statement 400.
```
25 CX=0: CO=0: SET COUNTERS FOR # OF GAMES WON
26 FOR G=1 TO 10:W=0

400 NEXT G
410 IF CX > CO THEN PRINT "X WINS THE SERIES": END
420 IF CO > CX THEN PRINT "O WINS THE SERIES": END
430 PRINT "SERIES IS TIED": END

5910 IF C$="X" THEN CX=CX+1
5920 IF C$="O" THEN CO=CO+1
```

Index

80-column boards, 307
6502 microprocessors, 5, 15, 18, 306
←, →, ↓, ↑, (arrow keys), 10-12, 72-74
: (colon), 58
, (comma), 44, 52, 55, 166, 167, 170
$ (dollar sign), 41, 42, 108
= (equal sign), 64
"(", ")" (parentheses), 48, 76, 106-107, 141
+ (plus sign), 192
] (prompt; square right bracket), 8, 43
? (question mark), 53
; (semicolon), 52, 54

ABS function, 78-79
Accounting software, 300-301
Accumulator variables, 95
Address directory program, 168-170
Addresses, 207
  PEEK and POKE for, 208-209, 218
Alphabetical order, 188-194, 222-223
AND connector, 68
Animation, 272-276
  man rowing, 276-277
  sound with, 283
  see also Computer graphics
APPEND instruction, 168-169
Apple II Computers, 3
  6502 microprocessor on, 5
  CP/M on, 306
  disk drives for, 21-25
  editing program lines on, 72-73
  insides of, 13-21
  keyboards for, 7-13
  speakers on, 280
Apple IIe Computers
  cursor control keys on, 72-73
  insides of, 17-20
  keyboard features on, 12-13
  memory on, 208
Apple II Plus Computers
  cursor control keys on, 72-73
  insides of, 14-17
  keyboard features on, 12
  memory on, 208
Apple DOS, *see* DOS
Applesoft BASIC (language), 28
  arrays (subscripted variables) in, 106-107, 113
  variables in, 40-42
  see also BASIC
Application programs
  artificial intelligence game, 197-202
  to calculate mortgage payments, 103-106
  to draw bar charts, 252-255
  to draw line graphs, 260-262
  for formatting of outputs, 228-230
  graphic of man rowing, 276-277
  homework assignment file, 171-173
  for inventory management, 186
  investment analysis system, 127-140
  to name state capitals, 70-71, 116-117
  to play roulette, 114-116, 154-157
  for return on investments, 120-123
  "Row, Row, Row Your Boat," 283-284
  song medley, 284-287
  telephone directory, 168-170, 187-197
  tic tac toe, 246-252
  to track monthly bills, 174-182
  word guessing game, 226-228
  for word processing, 235-237
Arithmetic
  BASIC functions for, 76-83
  program efficiency in, 159-160
  programming of, 47-52
Arrays (subscripted variables), 107-110
  assignment values to, 110-114
Arrow keys (←, →, ↓, ↑), 10-12, 72-74
Art, computer, 245-247, 259
ASC function, 193, 223
ASCII codes (American Standard Code for Information Interchange), 165
  list of, 202-204
Assembly language, 280
Assignment statements, 42
ATN function, 82

Backspace (←) key, 10-11, 74
Backups, 34-36, 75
Bar charts, 252-255
BASIC (language), 5, 28-29
  arithmetic in, 47-52
  arrays (subscripted variables) in, 106-107, 113
  conditional branch statements in, 60-68
  debugging programs in, 140-145
  different versions of, 306
  displaying and inputting data in, 52-58
  DOS commands and, 164-166
  editing program lines in, 72-75
  functions in, 76-83
  interpreters for, 159
  printing in, 43-47
  program structuring in, 85-89
  sound generation in, 279-280

variables in, 39-43
see also Programs
Battery packs, 307
Baud rate, 303
Bell, 11, 279
Binary files, 262, 268
Binary number system, 216-218
Binary search techniques, 131
Bits, 21, 207, 262
Blank spaces, 52-53
BLOAD command, 268
Booting the disk (bootstrapping), 23-24
Branches, *see* Conditional branch statements
BSAVE command, 268
Buffers, 305-306
Bugs (errors), 140
see also Errors
Byte codes, 265
Bytes, 21, 185, 207, 208, 262

CALL statement, 281
CAPS LOCK key, 12
Cassette recorders, 6, 7, 17, 20
CATALOG command, 36, 164, 165
Characters
ASCII codes for, 165, 202-204
see also Strings
Charts, 252-256
Chips, 15-16, 18-19, 306
CHR$ function, 165, 204, 223, 279
Circles, drawing, 257-259
CLEAR command, 114
Clocks, real-time, 307
CLOSE command, 167-168
Colons (:), 58
Color, 239, 241, 255
COLOR instruction, 241
Commands, 28
for binary files, 268
to control display, 123-125
DOS, 34-37, 164-166
DRAW and XDRAW, 267, 270, 271
Commas (,), 44, 52, 55
in data files, 170
as delimiters, 166, 167
Communications, 303-304
Computer Aided Design (CAD), 301
Computer graphics, 239
animation, 272-276
application program using, 276-277
art, 245, 247, 259
bar chart program for, 252-256
forming and saving shapes for, 262-272
high resolution, 255-259
line graph program for, 260-262
low resolution, 241-244
plotters and graphics tablets for, 307
printer capabilities for, 305
software available for, 301
sound with, 283, 286
tic tac toe program using, 246-252
Computers, 4-7, 28, 39
communications between, 303-304
Concatenation, 192, 221
Conditional branch statements
IF . . . THEN, 61-68
increasing efficiency of, 160
ON . . . GOSUB, 154
ON . . . GOTO, 126-127
Constants, 40, 41
CONT (CONTINUE) command, 97, 98
CONTROL-C, 68, 96-97
CONTROL-D, 165
CONTROL-G (bell), 11, 279
CONTROL (CONT) key, 7, 11
CONTROL-S, 97
CONTROL-X, 11
Copying
of diskettes, 34-36
of program lines, 73
COS function, 82
COT function, 82
CP/M (operating system), 289, 306
CSC function, 82
Cursors, 8
keys for control of, 10-12, 72-74
PEEK and POKE commands for, 209-210
statements for control of, 123-124
used in spread sheets, 292-294

Data
displaying and inputting, 52-58
tabular, 106-117
to test programs, 142-143
transmission of, 303
Data base packages, 299-300
Data files, 163-166
random access, 184-186
sequential, 168-173
Data items (data elements; fields), 184, 185
DATA statement, 56-58
for values in arrays, 111, 112
Debugging and testing programs, 140-145
Decimal number system, 216, 217
Deferred execution mode, 46
DEF FN instruction, 149-150
Delays of execution, 99-101
DELETE command, 37, 169
DELETE key, 12
Delimiters, 166, 167, 185
DIM (dimension) statement, 109-110, 113-114
Discount factors, 121, 130
Discount rate (hurdle rate), 120, 130
Disk drive controller boards, 16, 19, 21
Disk drives, 6-7, 16, 19, 21-25, 34-36

Diskettes (floppy disks), 6-7, 21-24
copying, 34-36
erasing files on, 36-37
loading programs from, 33-34
reading data files from, 169-170
saving programs on, 32-33
saving shapes on, 268
Disk Operating System, *see* DOS
Displays
80-column boards for, 307
of computer art, 245-247
of data, 52-54
freezing outputs on, 96-99
PEEK and POKE commands and, 209-210
SPEED command for, 127
statements and commands for, 123-125
text mode for, 239-241
see also Computer graphics
Documentation, 290
Dollar sign ($), 41, 42, 108
DOS (Disk Operating System), 23, 32, 34-38, 289, 306
commands for, 164-166, 182-184
sequential files and, 167-168
Doubly-subscripted variables, 108
Down arrow (↓) key, 12
DRAW command, 267, 270

Editing program lines, 72-75
Educational software, 301-302
END instruction, 43
Equal sign (=), 64
Erasing files, 36-37
Error messages, 9-10, 28, 140-141
for changes in size of arrays, 113
list of, 146-147
editing program lines to correct, 72-75
line numbers for correction of, 44
in programs, correction of, 30-31
testing and debugging of, 140-145
trapping of, 211-216
typing, correction of, 10-11
in use of READ and DATA statements, 58
ESCAPE sequences, 231
ESC (escape) key, 11-12, 72-74
ESC-P, 124
Event trapping, 210-211
EXEC command, 182-184
Execution of programs, 31-32
deferred and immediate modes for, 46
errors in, 141
loops for delays in, 99-101
speeding up, 159
STOP statement in, 97
EXP function, 82
Exponential format, 40
Exponential functions, 82-83
Exponentiation, 49-50, 76-77

Fields (data items; data elements), 184, 185
Files
binary, 262, 268
data, 163-166
data base packages for, 299-300
erasing, renaming, locking and unlocking, 36-37
random access, 184-186
sequential access, 166-173
sequential data (text), 182-183
FLASH command, 124, 125
Floppy disks, *see* Diskettes
Flowcharts, 89-90
Form letters, 231-234
FOR . . . NEXT statements, 93-95
for delay of execution, 100-101
increasing efficiency of, 161-162
Function keys, 12
Functions
Arithmetic, 76-83
ASC, 193
CHR$, 165, 204
in spread sheets, 295
for strings, 223-225
user-defined, 149-150
in word processing programs, 298

Games
artificial intelligence program, 197-202
paddles and joysticks for, 307
roulette, 114-116, 154-157
tic tac toe, 246-252
word guessing, 226-228
GET command, 125-126
GOSUB instruction, 151-152, 154
increasing efficiency of, 160
GOTO instruction, 60, 61, 63
conditional (ON . . . GOTO), 126-127
increasing efficiency of, 160
Graphics, *see* Computer graphics
Graphics tablets, 307
Graphs, 260-262
GR command, 239, 241

HCOLOR instruction, 255, 267
Hexadecimal number system, 217-218
HGR2 command, 273, 274
HGRA instruction, 256
HGR instruction, 239, 267, 272-274
High resolution graphics, 255-259
animation using, 272-277
forming and saving shapes in, 262-272
HLIN instruction, 243
HOME command, 10, 98, 123, 231
HPLOT instruction, 255-257
HTAB statement, 123, 124
Hurdle rate (discount rate), 120, 130

IF . . . THEN statement, 61-68
increasing efficiency of, 160
Immediate mode, 46
Index variables, 162
Infinite loops, 68
Initializing of diskettes, 24
Input devices, 6
Input/output (I/O) operations, 6
Inputs
of data, 54-58
GET command for, 125-126
INPUT statement, 28-29, 54-56, 65-67
for reading data files, 170
for values in arrays, 111
Instructions and instruction sets, 5, 39
for graphics, 243-244, 255-257
for manipulating strings, 223-225
Integer BASIC (language), 306
Integer variables, 41
Interface boards, 16, 19, 231, 307
Internal rates of return, 128, 130-132
Interpreters, 159
INT function, 77-78, 80-81, 99, 228
INVERSE command, 124, 125

Joysticks, 16-17, 20, 307

Keyboards, 7-13
event trapping and, 210-211

Languages, 306
assembly, 280
BASIC, 28
machine, 208
LEFT$ instruction, 224
Left arrow (←) key, 10-11, 74
LEN instruction, 223
LET statement, 42
Line graphs, 260-262
Line numbers, 29, 43-44, 46
in added subroutines, 184
addresses of, 215
in GOTO statements, 60
Lines, drawing, 243, 263-266
Linked lists, 188-192
LIST command, 30, 73, 75, 141
to write into files, 167, 182
Literals (string constants), 41
LOAD command, 33
Loading programs from diskettes, 33-34
LOCK command, 37
Locking and unlocking files, 37
Logarithmic functions, 82-83
LOG function, 82-83
Logic errors, 141-145
LOMEM instruction, 272
Loops, 88
to assign values to array members, 110-113
to delay execution, 99-101
FOR . . . NEXT statements in, 93-95
freezing display of, 96-99
increasing efficiency of, 161-162
infinite, 68
IF . . . THEN statements in, 64
sound in, 279, 280
Low-resolution graphics, 239-244

Machine language, 208
to generate sound, 280-283
Main memory, 5
Margins, for text window, 209, 210
MBASIC (language), 306
Memory, 5-6
addressing of, 207-208
on chips, 15, 18-19
for high resolution graphics, 272
machine language programs, 281
needed for programs, 289
PEEK and POKE for, 208-209, 218, 266
RAM, 21
saving binary files in, 268
Menus, 125-127
Microcomputers, 5
as word processors, 230
Microprocessing units (MPUs), 5, 15, 18
MID$ instruction, 224
Modems (modulator-demodulators), 303
Monitors (operating systems), 23
see also DOS; Operating systems
Monitors (video display units), 6, 17, 20, 239
Motherboards, 15, 18
MPUs (microprocessing units), 5, 15, 18
Music boards, 307

NEW command, 30
NEXT statement, 93-95, 161-162
NORMAL command, 124, 125
NOTRACE command, 145
Numbers, variables and constants, 40-41
Number systems, 216-218

ONERR GOTO statement, 212, 214-216
ON . . . GOSUB statement, 154
ON . . . GOTO statement, 126-127
Open Apple function key, 12
OPEN command, 167
Operating systems (monitors), 23, 289, 306
see also DOS
OR connector, 68
Output devices, 6
Outputs
freezing on screen, 96-99
subprogram for formatting, 228-230

Paddles, 16-17, 20, 307
Parameters, 209, 267

Parentheses
for functions, 76
in order of operations, 48
for subscripted variables, 106-107
syntax errors in, 141
Payback periods, 127-129
PEEK command, 208-210, 280
Peripherals, 307
interface boards for, 16, 19
see also Disk drives; Printers
Plotters, 307
Plus sign (+), 192
Pointers, 188-192
in artificial intelligence game, 198-199
POKE command, 75, 208-211, 281
graphics uses of, 266, 267, 274
Powers (mathematical), 76-77
PR# instruction, 231
Present Value method, 121, 129-131
Print buffers, 305-306
Printers, 6, 16, 19, 304-306
graphics software and, 301
printing instructions for, 45
programmed control of, 165-166
for word processing, 231
Printing
in programs, 43-47
types of printers for, 304-305
PRINT instruction, 28, 44-46, 52-54
CHR$ function used with, 204
CONTROL-D (CHR$(4)) with, 165
for data files, 167, 169
directed to printers, 231
in testing and debugging, 142-143
Print zones, 44-45, 52
Program files, 163
Programs, 5, 27-29, 39
arithmetic in, 47-52
arithmetic functions for, 76-83
attaching subroutines to, 182-184
available for purchase, 289-290
conditional branch statements in, 60-68
control of displays in, 123-125
debugging and testing of, 140-145
displaying and inputting data in, 52-58
DOS commands in, 164-166
editing lines in, 72-75
entering and correcting, 29-31
error trapping in, 211-216
executing, 31-32
flowcharts of, 89-90
increasing efficiency of, 158-162
loading from diskettes, 33-34
menus in, 125-127
printing in, 43-47
saving on diskettes, 32-33
sound generation in, 279-283
structuring of solutions for, 85-89
subroutines in, 151-157
user-defined functions in, 149-150
see also Application programs; BASIC; Software
Prompt (]), 8, 43

Question mark (?), 53

RAM (random access memory), 6, 15, 18-19, 21, 30, 207-208
Random access files, 164, 184-186
in artificial intelligence game, 198
in phone directory program, 187-197
Random number generator, 79-81
see also RND function
Rates of return, 120, 121, 130-131
Read only memory (ROM), 6, 15, 19, 23
READ statements, 56-58
Real-time clocks, 307
Real variables, 41
Record numbers, 185
Records, 184-185
REM (remark) statements, 58
RENAME command, 37, 181-182
Renaming of files, 37
REPEAT (REPT) key, 12, 72
RESET key, 7-8
RESTORE statement, 57
RESUME statement, 214
RETURN instruction, 151-152
RETURN key, 9, 11, 73, 74, 221, 231
ASCII code for, 166
RF modulators, 17, 20
RIGHT$ instruction, 224
Right arrow (→) key, 11, 73
RND function, 79-81, 99
in computer art, 245
in roulette program, 114
ROM (read only memory), 6, 15, 19, 23
ROT parameter, 267
RUN command, 5, 31

SAVE command, 32
Saving
of graphics, 262-272
of programs on diskettes, 32-33
of spread sheets, 296
SCALE parameter, 267
Screens, *see* Displays
SCRN instruction, 244
Scrolling, 9
SEC function, 82
Secondary storage, 6
Semicolons (;), 52, 54
Sequential access files, 163-164, 166-173
in application program, 174-182
Sequential data (text) files, 182-183
SGN function, 78

Shapes, drawing and saving, 262-272
Shape tables, 262-269
SHIFT keys, 8, 12
SIN function, 82
Software, 5
  for accounting, 300-301
  available for purchase, 289-290
  compatibility of, 18
  data base packages, 299-300
  educational, 301-302
  graphics, 301
  languages and operating systems, 306
  spread sheets, 290-296
  word processing programs, 297-299
  see also Application programs; Programs
Solid Apple function key, 12
Sound, 279-280
  in machine language programs, 280-283
  "Row, Row, Row Your Boat" program, 283-284
  song medley program, 284-287
SPC phrase, 53, 231
Speakers, 280-281
Special symbols, 13
Speech synthesizers, 307
SPEED command, 127, 279
Spelling correction programs, 298
Spread sheets, 290-296
SQR function, 76-77
Statements
  assignment, 42
  conditional branch, 60-68, 126-127, 154, 160
  to control display, 123-125
  multiple, on single line, 58, 158
STOP statement, 97
Storage
  secondary, 6
  sequential files in, 166
  see also Memory
Strings
  in arrays, 108, 111
  ASCII codes for, 165
  inputting of, 55, 56
  manipulation of, 221-225
  variables and constants, 41-42
STR$ instruction, 224-225
Subroutines (subprograms), 151-157
  in application programs, 177
  attaching to other programs, 182-184
  for formatting of outputs, 228-230
  increasing efficiency of, 160
  for sound generation, 281
Subscripted variables, 106-110
  assigning values to, 110-114
SYNTAX ERROR message, 28, 140, 146
Syntax errors, 140-141
System Master Diskette, 23, 34-35

Tabbing, 53-54
TAB command, 53-54, 231
TAB key, 12
Tabular data, 106-117
TAN function, 82
Television sets, 6, 17, 20, 239
Testing and debugging programs, 140-145
Text (sequential data) files, 182-183
Text mode, 239-241
TEXT statement, 75, 210, 239, 258
Text window, 209-210
TRACE command, 144-145, 166
Trigonometric functions, 81-82
Two-dimensional arrays, 108

UNLOCK command, 37
Unlocking files, 37
Up arrow (↑) key, 12, 74
User-defined functions, 149-150

VAL instruction, 225
Variables, 29, 39-43
  accumulator, 95
  INPUT and DATA statements for, 54-58
  in program efficiency, 159
  subscripted, 106-114
Video display units, 6, 17, 20, 239
VISICALC, 290-296
VLIN instruction, 243
VTAB statement, 123, 124

Word processing, 230-231
  application program for, 235-237
  of form letters, 231-234
  programs available for, 297-299
Wraparound, 9, 30
WRITE command, 167

XDRAW command, 270, 271

Z80 microprocessor chips, 306

APPLESOFT BASIC AND DOS REFERENCE GUIDE

ABBREVIATIONS: [ ] indicates optional item
nvariable indicates a numeric variable
svariable indicates a string variable

A. Immediate Mode

Any BASIC statement may be used in immediate mode. In addition, the following commands may be used in immediate mode:

BLOAD	Load a binary file into RAM, optionally specify address at which to load.	**BLOAD filename [,Aaddress]**
BSAVE	Save a binary file which starts at **address** and is **length** bytes long.	**BSAVE filename, Aaddress,Llength**
CATALOG	To display names of all files on the diskette in the indicated drive. If no drive is specified the current drive is assumed.	**CATALOG [Ddrive#]**
CONT	Continue a program interrupted by STOP or by CONTROL-C.	**CONT**
DEL	To delete program lines.	**DEL line1,line2**
DELETE	Erase specified file.	**DELETE filename**
EXEC	Load a text file into RAM without destroying current program. Has same effect as typing in contents of text file at keyboard.	**EXEC filename**
LIST	List lines of a program.	**LIST [line[,[line]]]**
LOAD	Load a program from diskette.	**LOAD filename**
LOCK	Write protect a file on diskette.	**LOCK filename**
NEW	Erase current program from RAM.	**NEW**
NOTRACE	Disable the Trace function.	**NOTRACE**
RENAME	Rename **filename1** on current drive to **filename2**.	**NAME filename1, filename2**
RUN	Execute program in RAM, optionally load and execute **filename**.	**RUN [filename]**
SAVE	Save current program diskette.	**SAVE filespec**
TRACE	Display numbers of lines executed. (Trace On)	**TRACE**
UNLOCK	Remove write protection from **filename**.	**UNLOCK filename**

B. Arithmetic Manipulation

ABS	Return the absolute value of X.	**nvariable= ABS(X)**
ATN	Return the arctangent of X.	**nvariable= ATN(X)**
CLEAR	Set all numeric variables to 0, all string variables to null; eliminates all arrays.	**CLEAR**
COS	Return the cosine of X.	**nvariable= COS(X)**
DEF FN	Define user function.	**DEF FNname(X)= expression in X**
DIM	Define size of an array.	**DIM var(subscripts) [,var(subscripts)]**
EXP	Return the exponential of X.	**nvariable= EXP(X)**
INT	Returns the largest integer less than or equal to x.	**nvariable= INT(x)**
LET	Assign values to variables. If assigning a value to a string variable, expression must be enclosed in quotes.	**[LET] variable= expression**
LOG	Returns the natural logarithm of X.	**nvariable= LOG(X)**
RND	Choose a random number between 0 and 1.	**nvariable= RND(1)**
SGN	Returns the sign of X (+1 if positive; −1 if negative; 0 if X=0)	**nvariable= SGN(X)**

SIN	Returns the sine of X.	**nvariable= SIN(X)**
SQR	Returns the square root of X.	**nvariable= SQR(X)**
TAN	Returns the tangent of X.	**nvariable= TAN(X)**
VAL	Returns the value of the number determined by the initial segment of the string **X$**.	**nvariable= VAL(X$)**

C. String Functions

ASC	Return the ASCII code of the first character of **X$**.	**nvariable= ASC(X$)**
CHR$	Return the character having ASCII code **n**.	**svariable= CHR$(n)**
LEFT$	Returns the leftmost **n** characters of **X$**.	**svariable= LEFT$(X$,n)**
LEN	Returns the length of **X$**.	**nvariable= LEN(X$)**
MID$	Returns a string of length **m** beginning with the nth character of **X$**.	**svariable= MID$(X$,n,m)**
RIGHT$	Returns the **n** rightmost characters of **X$**.	**svariable= RIGHT$(X$,n)**
STR$	Converts the number **X** into a string.	**svariable= STR$(X)**

D. Input/Output Statements

DATA	Identifies data items read by READ statements.	**DATA constant [,constant]...**
FLASH	Causes foreground and background colors to alternate repeatedly.	**FLASH**
HTAB	Sets the horizontal location (column) at which the cursor appears. Columns are numbered 1 at the left to 40 at the right of the screen (works only if followed immediately by PRINT or INPUT.)	**HTAB(n)**
INPUT	Reads data typed on the keyboard.	**INPUT[;]["prompt string";] variable variable [,variable]**
INVERSE	Causes foreground and background colors to be reversed.	**INVERSE**
GET	Reads the first character typed at the keyboard. Can input only a single character or digit; does not require hitting the RETURN key; does not echo input on screen.	**GET variable**
NORMAL	Causes screen display to return to normal foreground and background colors.	**NORMAL**
PRINT	Display data on screen.	**PRINT list of expressions**
READ	Assign values to variables using data items in DATA statements.	**READ variable [,variable]...**
RESTORE	Restore DATA pointer to first DATA statement.	**RESTORE**
SPC	Insert **n** spaces before printing.	**PRINT SPC(n)**
SPEED	Changes the speed with which output is displayed on the screen. Speeds can range from 0 to 255 with 0 the slowest.	**SPEED=n**
TAB	Move to column **n** before printing.	**PRINT TAB(n)**
VTAB	Sets the vertical location (row) at which the cursor appears. Rows are numbered 1 at the top to 24 at the bottom of the screen.	**VTAB(n)**

E. Program Control

END	End execution of current program.	**END**
FOR... NEXT	Repeat statements between FOR and NEXT as long as the variable is less than or equal to **Y**. The value of variable starts at **X** and with each repetition is increased by **Z**.	**FOR variable=X TO Y [STEP Z] ...NEXT [variable] [,variable]...**

GOSUB... RETURN	Transfer execution to line. Upon reaching RETURN, go to the line after the one containing GOSUB.	**GOSUB line ...RETURN**
GOTO	Transfer control to indicated line.	**GOTO line**
IF... THEN	If **expr** is true, then execute **clause1** (or GOTO line).	**If expr THEN clause1** **If expr GOTO line**
ON... GOSUB	Round the value of **expr** to an integer **n**. Go to the subroutine beginning at the nth listed line number.	**On expr GOSUB line [,line]...**
ON... GOTO	Round the value of expression to an integer **n**. Transfer control to the nth listed line number.	**ON expression GOTO line[,line]...**
STOP	Stop program execution.	**STOP**

F. Error Trapping and Miscellaneous Statements

ONERR	On error, transfer control to line.	**ONERR GOTO line**
POS	Returns the horizontal position of the cursor.	**POS(1)**
REM	Remark. Rest of the line is ignored.	**REM remark**
RESUME	Clear error condition and transfer control to line after error line.	**RESUME**

G. File Manipulation

To use a file manipulation or other DOS command within a BASIC program, the command must be included in a PRINT statement. **PRINT CHR$(4);"command"**

APPEND	Add to the end of a sequential file.	**APPEND filename**
CLOSE	Close a file.	**CLOSE filename**
OPEN	Open a file. For random files the record length in bytes (characters) must be specified.	**sequential files: OPEN filename** **random files: OPEN filename,Llength**
READ	Prepare a file for reading. All subsequent INPUT and GET statements will access the file until a CONTROL-D (CHR$(4)) is encountered. For random files, the record number at which reading begins must be specified.	**sequential files: READ filename** **random files: READ filename, Rrecord# [,Bbyte#]**
WRITE	Prepare a file for writing. All subsequent PRINT statements will write to the file until a CONTROL-D (CHR$(4)) is encountered. For random files, the record number at which writing begins must be specified. Insert commas between data items in PRINT statements.	**sequential files: WRITE filename** **random files: WRITE filename, Rrecord# [,Bbyte#]**

H. Graphics

COLOR	Define current background color for plotting (low-res) where **n** is color number.	**COLOR=n**
DRAW	To draw the specified shape number at column **c** and row **r** (hi-res).	**DRAW shape# AT c,r**
GR	Enter low-resolution graphics with 4 lines of text.	**GR**
HGR	Enter high resolution graphics with 4 lines of text.	**HGR**
HLIN	Draw a horizontal line at row **r** from column **c1** to **c2** (low-res).	**HLIN c1,c2 AT r**
HPLOT	Plot a point at column **c**, row **r**; or Draw a straight line from last plotting point to column **c2**, row **r2**; or Draw a straight line from column **c1**, **r1** to column **c2**, row **r2** (hi-res).	**HPLOT c,r** **HPLOT TO c2,r2** **HPLOT c1,r1 TO c2,r2**
PLOT	Plot a point at column **c**, row **r** (low-res).	**PLOT c,r**
ROT	Rotate shape when Drawing or Xdrawing, n=0 to 255 (hi-res).	**ROT=n**

SCALE	Scale shape when Drawing or Xdrawing, n=1 to 255 (hi-res).	**SCALE=n**
SCRN	Returns color of point plotted at column **c**, row **r**.	**SCRN (c,r)**
TEXT	Return to TEXT mode from graphics.	**TEXT**
VLIN	Draw a vertical line at column **c** from row **r1** to **r2** (low-res).	**VLIN r1,r2 AT c**
XDRAW	First request draws the shape in specified color, second request draws in background color; repeated requests alternate between specified color and background.	**XDRAW shape# at c,r**

I. Machine Language Interface

CALL	Call the machine language subroutine located at **address**.	**CALL address**
PEEK	Returns the contents of the indicated memory location.	**PEEK address**
POKE	Stores the indicated **value** in the specified memory location.	**POKE address, value**

Pre-publication reviewers say:

"This book is more accessible and informative for the novice than the Applesoft tutorial manual . . ."

"Beginning Apple users will accept this book eagerly . . . it's very impressive!"

Applesoft BASIC for the Apple II and IIe

Lois Graff and Larry Joel Goldstein

Now—a complete and practical guide to BASIC programming for the Apple II and IIe, using Applesoft BASIC! Designed for novices and experienced users alike, this unique guide walks you through the fundamentals of BASIC. It also offers a step-by-step introduction to structured programming and discusses a wide variety of practical business applications. You'll also find a convenient tear-out reference guide inside that concisely summarizes the BASIC and DOS commands presented throughout the text!

In this book you will also find:

- a clear, concise outline of how the Apple II/IIe computer really works
- helpful tips on easing the frustrations of programming
- immediate applications to business, graphics, games, word processing, sound, and more!

CONTENTS

A First Look at Computers/Running Programs on Your Apple II/IIe Computer/Beginning BASIC/Structuring a Solution/More BASIC/Organizing a Program/Using Your Computer as a File Cabinet/A Peek Into Your Apple's Memory/Playing with Words/Playing with Pictures/Playing with Sounds/Software You Can Buy/Expanding the System

ISBN 0-89303-320-0

GRAFF
GOLDSTEIN

APPLESOFT BASIC for the Apple II & IIe

Brady